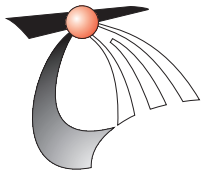
Design and Analysis of Transport Protocols for Reliable High-Speed Communications



CTIT Ph.D-thesis series No. 97-15

P.O. Box 217 — 7500 AE Enschede — The Netherlands
telephone +31-53-4893779/4892100 / fax +31-53-4894524

Centre for
Telematics and
Information
Technology

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Oláh, András László

Design and Analysis of Transport Protocols for Reliable High-Speed Communications / András László Oláh. - Ill. - (CTIT Ph.D-thesis series, ISSN 1381-3617; no. 97-15)
Thesis University of Twente, Enschede. - With ref. - With summary in Dutch and Hungarian.
ISBN 90-9010872-6
Subject headings: computer networks / transport protocols / communication system planning.

Design and Analysis of Transport Protocols for Reliable High-Speed Communications

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector Magnificus,
prof.dr. F.A. van Vught,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 28 augustus 1997 te 16.45 uur

door

András László Oláh
geboren op 28 november 1966
te Boedapest, Hongarije

Dit proefschrift is goedgekeurd door de promotor
Prof.dr.ir. I.G.M.M. Niemegeers

dolgozni csak pontosan, szépen,
ahogy a csillag megy az égen,
ugy érdemes.

József Attila, 1935–1937

Summary

The design and analysis of transport protocols for reliable communications constitutes the topic of this dissertation. These transport protocols guarantee the sequenced and complete delivery of user data over networks which may lose, duplicate and reorder packets. Reliable transport services are required by a wide range of applications such as the World-Wide Web, remote network access, and distributed computing.

The design of these protocols is heavily influenced by the parameters of the underlying network infrastructure and by the assumptions about the host computers and applications. Therefore the recent advances in optical transmission and computer technologies stimulated the design of several novel transport protocols. Many of the proposed protocols use similar or at least related techniques. Our goal with this thesis is to *improve the understanding* of reliable communications by analyzing the protocols that implement this service and to *contribute to the design* of reliable transport protocols.

The basis of our analysis is the formal specification and verification of the protocol mechanisms under investigation. The behavior of the protocol is captured by a state-transition system and properties are established using assertional reasoning. The framework is capable to handle unbounded and modulo-N state variables and to capture real-time aspects of the protocols which is essential for the modeling of realistic systems. Practical protocols of considerable complexity are specified and verified in the thesis.

One advantage of the formal verification is that it increases our confidence in the correctness of these protocols. The formalism forces us to clarify all the details of the working of the protocol and to state explicitly every assumption about the protocol and its environment. During the process of the verification one also gains insight into the mechanisms of the protocol. But probably the most important result is that during the verification we obtain conditions for the correctness of the protocol in the form of inequalities on some protocol parameters. These conditions allow the comparison of the different protocol mechanisms and can be used to judge the suitability of a protocol for a certain environment.

The functionality of transport protocols can be naturally divided into data transfer and connection management. Data transfer deals with the sequenced delivery of user data, while connection management is concerned with the orderly setup and release of connections.

In the thesis we study three different data transfer protocols. The usage of timestamps in data transfer protocols is analyzed in detail through the example of the PAWS mechanism which was proposed as an extension to TCP. The analysis reveals that the use of timestamps increases the functionality of the transport protocol by facilitating the simple measurement of round-trip delays, but it also reduces the maximum allowable transmission rate as compared to the plain sliding-window protocol.

Another data transfer protocol called SNR is analyzed which is based on the idea of periodic state exchange. We start from an earlier specification of SNR and compare it to the plain sliding-window protocol. The analysis reveals that the maximum transmission speed achievable by that SNR specification is higher than that of the plain sliding-window protocol, but it comes with a serious limitation. In the SNR specification it is assumed that no duplicates are generated by either the network or the transport protocol itself. This assumption may seriously limit the effective performance of the protocol in case of losses in the network and demonstrates the importance of considering all the assumptions when selecting a protocol for a certain environment.

The use of timestamps is also investigated in the context of connection management protocols. The detailed analysis of the connection setup protocol SCMP is presented which is based on the assumption that clocks of computers can be synchronized relatively cheaply even in a large network. In our verification it is proven that the safety of the protocol does not depend of the synchronization assumption, therefore the protocol can be used safely in cases when there are no absolute guarantees of the clocks being synchronized. Since practical clock synchronization algorithms give only probabilistic guarantees, our result provides an important theoretical support of the applicability of the protocol in practical environments.

Based on earlier work by others, a family of connection management protocols is analyzed that use a cache to store information needed to shorten the connection setup latency. We contribute to this work by proposing improvements which allow to reduce considerably the memory usage of these protocols. Furthermore, we show that the correctness of the protocol can be assured without assuming an upper bound on the incarnation lifetime, i.e., the maximum duration of a connection. This result greatly improves the practical applicability of the protocol.

Samenvatting

Het onderwerp van deze dissertatie is het ontwerpen en analyseren van transportprotocollen voor betrouwbare communicatie. Deze transportprotocollen garanderen het volledig en in de juiste volgorde afleveren van gebruikersdata door netwerken die pakketten kunnen verliezen, dupliceren of van volgorde verwisselen. Zulke betrouwbare diensten worden vereist in een breed scala aan toepassingen zoals het "Worldwide Web", gedistribueerd rekenen en toegang tot "remote" netwerken.

Het ontwerpen van deze protocollen is in grote mate afhankelijk van de parameters van de infrastructuur van het betreffende netwerk en de gemaakte aannames over de aangesloten computers en gebruikte toepassingen. Zo heeft de recente vooruitgang in optische transmissie en computertechnologie geleid tot menige nieuwe transportprotocollen. Vele van deze maken gebruik van verwante technieken. Het doel van dit proefschrift is het *vergroten van het inzicht* in betrouwbare communicatie door de analyse van de protocollen en *bij te dragen tot het ontwerp* van betrouwbare protocollen.

De formele specificatie en verificatie van de onderzochte protocolmechanismen vormt de basis van de analyse. Het gedrag van het protocol wordt beschreven door een toestandsovergangssysteem, waaruit door middel van "assertional reasoning" eigenschappen worden afgeleid. Binnen dit model kan er met onbegrensde en modulo-N toestandsvariabelen worden gewerkt en bovendien kunnen "real-time" aspecten van protocollen meegenomen worden. Dit is essentieel voor het modelleren van realistische systemen. In dit proefschrift worden practische protocollen van aanzienlijke complexiteit gespecificeerd en geverifieerd.

Een voordeel van formele verificatie is dat zij het geloof in de correctheid van de protocollen vergroot. Het gebruik van een formalisme dwingt af dat alle details van de werking van het protocol duidelijk worden en dat alle veronderstellingen betreffende het protocol en zijn omgeving expliciet worden gemaakt. Bovendien wordt tijdens de verificatie het inzicht in het protocol vergroot. Het belangrijkste resultaat is misschien wel dat de voorwaarden voor de correctheid van het protocol uitgedrukt blijken te kunnen worden in ongelijkheden in enkele protocolparameters. Deze voorwaarden maken het mogelijk dat verschillende protocolmechanismes kunnen worden vergeleken en gebruikt kunnen worden om de geschiktheid van een protocol voor een bepaalde omgeving te beoordelen.

De functionaliteit van transportprotocollen kan op een natuurlijke manier verdeeld wor-

den in dataoverdracht en connectiemanagement. Dataoverdracht betreft het op volgorde afleveren van gebruikersdata terwijl connectiemanagement te maken heeft met het ordentelijk opzetten en weer afbreken van een verbinding.

Drie dataoverdrachtsprotocollen worden in dit proefschrift bestudeerd. Het gebruik van "timestamps" in deze protocollen wordt in detail geanalyseerd aan de hand van het PAWS-mechanisme dat als uitbreiding van TCP is voorgesteld. Uit die analyse komt naar voren dat het gebruik van timestamps de functionaliteit van het protocol vergroot door het vergemakkelijken van het meten van een "round-trip" vertraging. De maximaal toegestane transmissiesnelheid wordt echter verkleind in vergelijking met een gewone "sliding-window" protocol.

Een ander protocol, SNR, dat op het periodiek uitwisselen van toestanden is gebaseerd, wordt ook geanalyseerd. Eerst wordt een eerdere versie van SNR bekeken en vergeleken met een gewoon sliding-window protocol. Het resultaat van de analyse is dat de maximaal haalbare transmissiesnelheid van die SNR-versie hoger is dan het sliding-window protocol. Dit gaat echter samen met een ernstige beperking. De SNR-specificatie gaat er namelijk van uit dat noch het netwerk- noch het transportprotocol duplicaten genereert. Deze aanname zou de prestatie van het protocol in het geval van verliezen in het netwerk ernstig kunnen beperken, wat het belang aantoont van het meenemen van alle veronderstellingen bij de selectie van een protocol voor een bepaalde omgeving.

Het gebruik van timestamps wordt ook in de context van connectiemanagement protocollen onderzocht. Er wordt een gedetailleerde analyse van het opzetten van een verbinding in SCMP gepresenteerd. Het mechanisme veronderstelt dat de klokken van computers zelfs in een groot netwerk op een relatief goedkope wijze gesynchroniseerd kunnen worden. De verificatie toont aan dat de betrouwbaarheid van het protocol niet van deze veronderstelling afhangt en dat het protocol dus ook gebruikt kan worden in omgevingen waarin de mogelijkheid om klokken te synchroniseren niet absoluut gegarandeerd is. Gegeven het feit dat zo'n garantie praktisch niet te geven is, biedt dit resultaat een theoretische ondersteuning van het protocol in een praktische omgeving.

Uitgaande van eerder door anderen uitgevoerde analyses, wordt, ten slotte, een familie van connectiemanagement protocollen onderzocht. Deze familie wordt gekenmerkt door het feit dat de protocollen gebruik maken van een cache voor de opslag van informatie om zo de vertraging bij het opzetten van een verbinding te verkorten. De bijdrage van dit proefschrift hieraan betreft verbeteringen die tot een aanzienlijke vermindering van het geheugengebruik leiden. Bovendien wordt aangetoond dat de correctheid van het protocol kan worden gegarandeerd zonder te veronderstellen dat de maximale duur van een verbinding een bovengrens heeft. Ook dit resultaat vergroot de praktische toepasbaarheid van het protocol in sterke mate.

Összefoglalás

Ennek a disszertációnak a témáját a megbízható kommunikációra használható transzport protokollok tervezése és analízise adja. Az ilyen típusú transzport protokollok garantálják az adatok sorrendhelyes és hiánytalan átvitelét olyan hálózatok felett is, amelyek elveszíthetnek, megduplázhatnak vagy felcserélhetnek csomagokat. Számos alkalmazás működésének a feltétele a megbízható transzport szolgáltatások megléte. Ilyen alkalmazás például a World-Wide Web, a távoli hálózati hozzáférés, vagy az elosztott számítógépes rendszerek.

Ezen protokollok tervezését erősen befolyásolják az alapszolgáltatást nyújtó hálózat paraméterei és a kommunikációban résztvevő számítógépekről és alkalmazásokról tett egyéb feltevések. Ebből kifolyólag az utóbbi időben számos új transzport protokollt fejlesztettek ki az optikai átviteli technológiában és a számítógépgyártásban végbement fejlődés hatására. Ezek az új protokollok több ponton mutatnak hasonlóságot illetve használnak fel közös építőelemeket. Ezzel a disszertációval az a célom, hogy egyfelől javítsam a megbízható kommunikáció lényegének a megértését az ezen szolgáltatást megvalósító protokollok analízisén keresztül; másfelől pedig, hogy hozzájáruljak a megbízható transzport protokollok tervezéséhez.

A vizsgálódás alapját a protokollok formális specifikációja és verifikációja képezi. A protokollok működését egy állapotátmenet-rendszerrel modellezzük, és a rendszer tulajdonságait a temporális logika módszereivel bizonyítjuk. Ez a leírási módszer képes kezelni korlátlan és modulo-N állapotváltozókat, és a vizsgált rendszer valós idejű aspektusait is természetes módon lehet benne figyelembe venni. A verifikációs módszer ezen tulajdonságai elengedhetetlenül fontosak valóságos rendszerek modellezéséhez. A disszertációban jelentős komplexitású, a mindennapos gyakorlatban használt protokollok specifikációja és verifikációja is megtalálható.

A formális verifikáció egyik előnye, hogy bizonyíthatjuk a protokoll helyes működését. A formalizmus megköveteli, hogy a protokoll működésének minden részletét tisztázzuk és hogy a rendszerrel kapcsolatos minden feltevésünket explicit módon megfogalmazzuk. A verifikácó során a protokoll működési mechanizmusát is jobban megértjük. Azonban a verifikáció talán legfontosabb eredménye, hogy a protokoll helyes működésének feltételeihez jutunk a rendszer paraméterein értelmezett egyenlőtlenségek formájában. Ezen feltételek alapján az azonos célra szolgáló protokollok objektíven összehasonlíthatók és a feltételek annak megítélésére is alkalmasak, hogy a protokoll mennyire használható

egy adott környezetben.

A transzport protokollok funkcióit két természetesen adódó osztályba lehet sorolni: adatátvitel és kapcsolat-menedzsment. Az adatátvitel feladata az adatok hibátlan átvitele, míg a kapcsolat-menedzsment foglalkozik a kommunikáló partnerek között fennálló kapcsolatok felépítésével és bontásával.

A disszertációban három különböző adatátviteli protokollt elemzek. Részletesen is megvizsgálom az időbélyegek adatátviteli protokollokban való használatának kérdéseit a PAWS mechanizmus analízisén keresztül. A PAWS a széleskörűen használt TCP protokoll kiterjesztése. Az analízis során kiderül, hogy az időbélyegek használata kiterjeszti ugyan a protokoll funkcionalitását azzal, hogy elősegíti a késleltetések mérését, másfelől viszont csökkenti az elérhető maximális adatátviteli sebességet az ugyanilyen paraméterekkel működő hagyományos csúszóablak protokollhoz viszonyítva.

Egy másik adatátviteli protokoll, az ún. SNR, alapját az állapotinformációk periódikus és kölcsönös cseréje képezi. A disszertációban az SNR egy, az irodalomban korábban megjelent, specifikációját vetem össze a hagyományos csúszóablak protokolléval. Az analízis megmutatja, hogy az SNR által elérhető maximális adatátviteli sebesség magasabb, mint amit a csúszóblak protokollal elérhetünk. Ennek azonban komoly ára van a protokoll más tulajdonságai szempontjából. Az SNR-nek ebben a specifikációjában a szerzők feltesznik, hogy sem a hálózat, sem pedig a transzport protokoll maga nem duplázhat meg csomagokat. Ez a feltevés erősen korlátozhatja a protokoll effektív átviteli sebességét olyan esetekben, amikor a hálózatban elveszhetnek csomagok. Ez az effektus jól mutatja annak fontosságát, hogy egy protokoll minden jellemzőjét alaposan mérlegelnünk kell annak eldöntéséhez, hogy a protokoll megfelel-e egy adott környezetben.

Az időbélyegek használata a kapcsolat-menedzsment protokollok körében is felmerül. Az SCMP nevezetű kapcsolat-menedzsment protokoll részletes analízise található meg a dolgozatban. Ez a protokoll azon a feltevésen alapszik, hogy még nagy kiterjedésű hálózatokban is viszonylag könnyen szinkronizálhatóak a számítógépek órái. Bebizonyítom, hogy az SCMP protokoll biztonságosságának nem feltétele az órák szinkronizációja, tehát a protokoll olyan esetekben is alkalmazható, amikor nincs abszolút garancia arra, hogy az órák mindig szinkronban vannak. Mivel a gyakorlati szinkronizációs protokollok csak valószínűségi garanciákat adnak a szinkronizációra, a fent említett eredmény fontos elméleti támogatást nyújt az SCMP protokoll gyakorlati alkalmazhatóságához.

Mások korábbi munkáit is felhasználva a kapcsolat-menedzsment protokollok egy olyan családját is megvizsgálom, amelyek egy ún. cache-t használnak a protokoll információk tárolására és ezáltal a kapcsolatfelvétel késleltetésének csökkentésére. Számos olyan módosítással járulok hozzá ezen protokollok tervezéséhez, amelyek lehetővé teszik a protokoll memóriaigényének jelentős csökkentését. Továbbá azt is megmutatom, hogy ezen protokollok helyes működésének nem szükséges feltétele a kapcsolatok hosszának korlátozása. Ez az eredmény nagyban növeli a protokoll-család gyakorlati alkalmazhatóságát.

Acknowledgments

I have been dreaming about this moment of finishing my Ph.D. dissertation for a long time. Before I 'dot the *i*,' I would like to turn back and acknowledge the help and support I got from others.

The supervisor of my research, Sonia Heemstra fulfilled her tasks well beyond the official level. I greatly appreciate the freedom she gave me to find a topic that I was most interested in and her patience in reading and commenting on my papers which were sometimes rather far from her professional interest. Sonia and her partner Sabih have also turned to be very good friends in private life helping me and my family to find our way in a foreign land.

I also would like to thank my promotor Ignas Niemegeers for the possibility of being an 'AIO' at his group for four years. His help and guidance during the writing of the thesis is greatly appreciated.

Although at the moment of writing these words I have not met Prof. Shankar yet, I already owe him a lot. His work on protocol verification provided the basis of my research and his insightful remarks on earlier drafts of the thesis were invaluable.

The head of Traffic Lab., Miklós Boda has in a special way contributed to finishing my dissertation. Without his constant threats of 'public execution' for missing my numerous deadlines of completing the manuscript, I don't think I would even be close to this point now.

My colleagues at TIOS and now at Ericsson have created a pleasant and stimulating working environment and I take this opportunity to say them all a *Thank you!*

Contents

Chapter 1

Introduction

This thesis is devoted to the design and analysis of transport protocols for reliable communications. Reliable transport protocols guarantee the sequenced and complete delivery of user data over networks which may lose, duplicate or reorder packets. Such a transport service is required by a wide range of applications.

Many transport protocols have been designed in the past decades to implement reliable communications. Two major peaks can be distinguished in the scientific activity directed towards reliable transport protocols. The first was at around '75–'81 [Dal75, FW78, SD78, Tom75, Wat81], when the foundations of the current Internet [CK74] were laid. Protocols such as TCP and Delta-t originate from this period.

Another wave of interest came in the late eighties, early nineties, when it was realized that the design of transport protocols is affected by advances in networking and computing technologies in a number of ways. The relative cost of network bandwidth and computer memory decreased dramatically, the increasing role of latency became evident [DDK$^+$90, Kle92]. Several proposals for alternative ways to design reliable transport protocols were published in this period [BF93, Bra92, Bra94, Che88, JBB92, LSW91, NRS90, Pro92].

Our major goal with this thesis is to improve the understanding of reliable communications by analyzing the protocols used for implementing this service. The goal is achieved by the formal specification and analysis of the important protocol mechanisms. This analysis offers the following results:

- It improves our confidence in the correctness of the protocols.
- The formal analysis provides the explicit conditions for the correctness of these protocols. Being aware of these conditions is essential for planning complex communication systems.
- Pointing out the similarities and differences of the existing protocols improves the understanding of reliable communications which is useful for designers of future protocols.

The structure of this introduction is as follows. First we place reliable communications

into context by specifying its relation to other sorts of communications. Then the service provided by the network layer to transport protocols is characterized. In Section 1.3 a brief overview of transport protocol mechanisms is given. Section 1.4 discusses the contributions of this thesis.

1.1 Integrated services networks

Integrated services (IS) networks are a major topic of research nowadays. The ultimate goal of this research is to design and deploy a networking infrastructure which is capable to serve a wide range of applications. These applications range from traditional electronic mail or voice call to emerging multimedia applications. Therefore, IS networks are a replacement of conventional telephony, cable-TV, and data communications networks. The integration of these networks opens the way for more sophisticated information services.

The service expected by different applications from the underlying network can be described from several viewpoints:

- connectivity;
- reliability (qualitative specification of service);
- performance guarantees (quantitative specification of service).

1.1.1 Connectivity

In our terminology, *connectivity* is defined by the number of participants in the communication and their relation to each other. Therefore we can distinguish one-to-one, one-to-many, and many-to-many communications. Furthermore, the flow of information can be uni-directional or bi-directional.

A conventional two-party telephone call is an example of *one-to-one* communication where the flow of information is *bi-directional*. Remote login to a distant computer or sending a telefax message are further examples of one-to-one communications. On the contrary, a TV broadcast is a typical case of *one-to-many* communication where the flow of information is *uni-directional*.

Tele-learning, an application which is often mentioned among the many new applications made feasible by modern networking technology, requires a one-to-many but bi-directional connectivity. Most of the information is carried from the teacher to the students, but students also have the possibility to react.

1.1.2 Reliability

User information is transmitted in *data units* over a digital network. Applications have freedom in defining their unit of data. For a request-response type application, such as name resolution, the unit of data may be a variable-sized record capable to hold either a request or a response. A single character is the data unit for a remote login application; a frame can be the data unit for a video application.

Communication consists of the exchange of multiple data units. The usefulness of communication requires that the sequence of data units presented to the receiving user(s) is related in some way to the sequence of data units submitted by the sending user. The *reliability* of the service is the *qualitative* description of this relation. Less reliable service means less constraints on the stream of data units presented to the receiver. Note that the term reliability has another common meaning in a different context where it refers to fault tolerance. That sort of reliability is not addressed in this thesis.

The following criteria can be used to characterize the reliability of single source communications [DDK+90]:

Freedom from bit errors: Data units must not be altered while in transit and the network must not generate data. This definition assures that there is a causality relation between sending and receiving data, i.e. reception of a data unit must always be preceded by the sending of an identical data unit.

Completeness: Any data unit submitted by the sender is eventually delivered to the receiver.

Sequencing: Data units are delivered to the receiver in the same order as they were submitted by the sender. Strict sequencing also means freedom from duplicates.

Particular applications may require several combinations of the above requirements. The application used for name resolution[1] in the Internet requires only freedom from bit errors. Each name resolution request is sent in a separate message to a name server. The loss or duplication of a request does not matter, a repeated request will yield the same result. On the contrary, remote login requires sequenced, complete delivery which is free from bit errors.

The requirements of a real-time voice conversation, for example, can be described by sequenced delivery and freedom from bit errors. The loss of a few messages is tolerated because it causes less degradation of the quality than the varying delay caused by retransmissions. A sophisticated file-transfer application may require only completeness and freedom from bit errors without sequencing. The eventual re-sequencing of the data blocks could then be achieved as a side-effect of writing each block at its corresponding offset to the disk. More efficient protocol processing is the advantage of these relaxed ordering requirements [CT90].

[1] The mapping of host names to network addresses among others.

In the context of multicast communications when there may be multiple sources and/or receivers, the above characterization is not sufficient. The relations of data units produced by different senders must also be specified. Further information on the definition of reliability for multicast communications can be found in [ACC+94, CM84, Dio94, Lam78].

1.1.3 Performance parameters

Applications can also be classified according to their quantitative performance requirements. In this thesis the classification proposed in [BCS94] will be used:

Real-time applications are usually some sort of "playback" applications, i.e. the receiver tries to reconstruct the original signal from the information transmitted over the network. To compensate the variable delay of the network, the application may delay the replay of the signal. Data which arrives after its playback time is essentially useless in reconstructing the real-time signal. The quality observed by the user depends on two factors: loss rate and playback delay. Loss results in distortion, excessive delay hinders interactive communications (think of a telephone conversation or a distributed simulation). The amount of playback delay and data loss which may be tolerated depends heavily on the sort of application.

Elastic applications always wait for late data to arrive. The application typically uses the arriving data immediately, rather than buffering it for some later time, and will always choose to wait for the incoming data rather than proceed without it. These applications are also sensitive to delay, but in a different way. The performance of the application depends on the average delay, rather than on the delay of individual packets. Because arriving data can be used immediately, these applications do not require any *a priori* characterization of the service in order for the application to function.

1.1.4 Type of communication considered here

In this thesis, we focus our attention to protocols which provide *reliable, one-to-one* communications. By reliable we mean sequenced, complete delivery of data that is free from bit errors. This sort of reliability can only be achieved by retransmission-based protocols, which on the other hand makes it impossible to provide real-time guarantees such as bounded delay. Therefore, these protocols can serve *elastic applications* only.

The sort of service described above is required by "conventional" data communications, such as file-transfer or remote login. It does not mean, however, that the need for this sort of service will disappear in the future. The World-Wide Web [BLCL+94], for example, a flexible hypertext-based information system which is the main force behind the current success of the Internet, also requires this sort of service from the underlying protocols.

1.2 Operating environment of transport protocols

The functionality to provide reliable data transfer is implemented at the transport layer of the OSI Reference Model [Tan89]. The transport layer operates right on top of the network layer and it is the first end-to-end layer of the Reference Model. That is, the transport layer is present in the protocol stack of end systems of a network such as host computers, but not in intermediate nodes such as routers. The examination of the network layer, which provides the services available to the transport layer, brings us closer to the understanding of transport protocols.

1.2.1 Datagram networks

In traditional computer communications, a network layer which provides little guarantees proved to be a successful paradigm [Cla88]. The network layer protocol of the Internet, IP [Pos81a] supports a datagram service with no hard guarantees about the loss, delay, duplication or ordering of packets. If the application needs further guarantees about the delivery of data, it has to be implemented by the end-to-end protocols.

We mention two of the reasons listed by Clark [Cla88] in favor of this architecture:

Survivability in the face of failure is an important requirement from an internetwork. Since the intermediate nodes in a datagram network are essentially stateless, communication between end systems can continue in case of a failure as long as the network remains connected.

Easy integration of a variety of networking technologies is also required. Because the service that the participating networks must support is rather simple, a wide variety of networking technologies can be integrated with minimal effort.

The service provided by such a datagram network can be modeled as a collection of unreliable channels between any pair of hosts. We make very few assumptions about this service:

- freedom from bit errors;
- the existence of an upper bound on delay.

Both of these are fundamental requirements for the existence of *any* reliable transport protocol.

Freedom from bit errors is achieved by the use of error-detection or error-correction codes [Bla83]. Strictly speaking, such techniques can never guarantee that bit errors are always detected, but the probability of such events can made arbitrarily low by adding sufficient redundancy to the transmitted information. If a packet appears to be corrupted, it is simply discarded which makes this event equivalent to packet loss.

Most networks have only loosely defined mechanisms to limit the lifetime of packets. Any function that requires processing by routers decreases the achievable transmission speed

and/or increases the cost of routers, which is the reason for the lack of strict lifetime enforcement [Che89]. Future networks are not likely to change in this respect, either. It is, however, still possible to give an upper limit of the delay which is not likely to be exceeded. This limit in the current Internet is 120 seconds [Pos81b] which is well over the usual measured delays [SAGJ93]. This "over-engineering" assures that the probability of a packet delayed longer than the maximum packet lifetime is practically zero.

In practice, datagrams may be mis-routed, i.e., delivered to the wrong host. Mis-routing can always be detected if we assume freedom from bit errors because datagrams carry the identity of the destination in their header. Thus mis-routed packets are simply discarded. This justifies the existence of logical channels between hosts in our abstract model of the network service which will be presented below in Section 1.2.3.

1.2.2 Integrated services networks

Although the stateless model proved to be very successful in practice, the move towards integrated services networks demands the revision of this paradigm. The stateless model by definition excludes the possibility of resource reservations which is needed by real-time applications.

A related problem is the accounting of traffic generated by users [Cla88]. Accountability has not been a major concern in centrally founded research networks such as the Internet in the early days, but it becomes an important issue when the access to the network and the information available on the network is offered as a service by commercial entities.

Both resource reservation and accounting require the treatment of packets as part of a sequence from source to destination as opposed to independent datagrams. Asynchronous Transfer Mode (ATM) [HHS94, MS95] which is the technology base for implementing Broadband Integrated Services Digital Networks (B-ISDN) [DP93, Sta95], is based on the virtual circuit model. Each cell, the unit of data transmitted in an ATM network, belongs to a virtual circuit which has to be established before communication can start. Reservation of resources is part of the connection setup procedure.

Another approach is followed within the Internet community. IPng [DH95], the next generation Internet Protocol, retains the datagram model but provides the means to express the relationship between a sequence of datagrams. These sequences are called flows in the Internet terminology. Resource reservations can be made on the basis of flows, which requires the storage of per-flow state information in the routers. In order to preserve the robustness of the architecture, the transient nature of this state (so called *soft-state*) is emphasized in [Cla88, DH95].

It is generally agreed that IS networks should continue to provide support for elastic applications which do not need reservations. In the Internet architecture, this type of usage is still considered to be fundamental. Within the ATM-world, support for the so-called available bit-rate (ABR) class of traffic has been added recently [All95, Jai95].

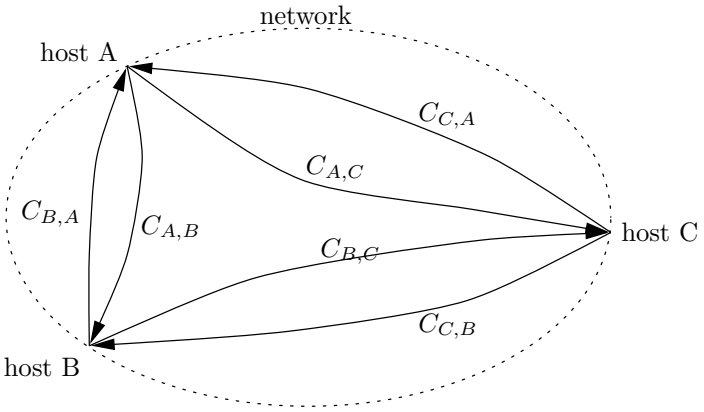


Figure 1.1: *Conceptual model of the available network service*

Our main interest is the type of service IS networks provide to elastic applications. The virtual circuit model of ATM networks allows to make stronger assumptions about the network service. Since the cells of a connection follow the same path, reordering of cells is excluded. The specification also excludes the possibility of duplicate cells.

In our view, however, it would not be wise to rely on these stronger assumptions in the design of transport protocols for a number of reasons:

Duplicates can be generated by the transport protocols themselves, not only in the network. The reliability of these protocols is based on the retransmission of packets that are believed to be lost. Although it is possible to design a transport protocol in such a way that it never generates duplicates, it may lead to a rather inefficient design [OHdG96a]. This issue will be further explained in Chapter 3.

Reordering in datagram networks is usually caused by multi-path routing [Hui95, Ste95]. Although cells in an ATM network follow the same route, ATM will not be the exclusive networking technology in the near future. While different parts of the network are based on ATM, these and other networks will still be connected by an internetworking layer. If the internetworking protocol is datagram based, then the possibility of reordering is still there.

1.2.3 Abstract model of network service

The service of a typical network layer can be modeled as shown in Figure 1.1. Hosts are connected by point-to-point uni-directional channels. There is a uni-directional logical channel between any pair of hosts; bi-directional communication is modeled by parallel channels in the opposite direction. Any channel may delay, lose, duplicate or reorder the packets in transit. Our only assumptions are that the delay has a known upper limit and that packets cannot be altered in the channels. A formalized description of the network service is given in Chapter 2.

Having specified the service required by the application and the service offered by the

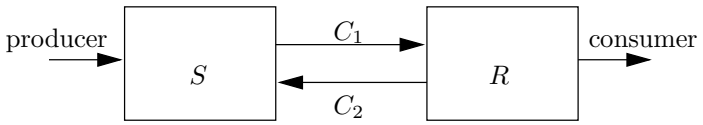


Figure 1.2: *Producer-consumer model of reliable data transfer*

network, we can turn our attention to the question: How does a transport protocol achieve the required service?

1.3 Reliable transport protocols

The communicating transport entities, hosts in short, have to maintain status information to cope with the unreliability of the network service. A *connection* is open when the status information is present. Starting with no status in the hosts, a connection has to be *opened* when either side has information to send. When the communication is over, the connection is *closed* and the status information is released. The exchange of the user-level information takes place while the connection is open.

Reliable transport protocols are usually split into two subproblems [Wat89, Sha91]: *connection management* and *data transfer*. Connection management deals with the setup and release of state in the entities. Data transfer deals with the sequenced, complete delivery of user data. The phases of transport protocols are not always strictly separated. A connection setup packet may already carry user data, for example. Also, the release of state can take place without exchange of any messages

1.3.1 Data transfer

Data-transfer protocols can be treated as a producer-consumer problem shown in Figure 1.2. The system consists of the producer (or sender) S and the consumer (or receiver) R, which are connected by two unreliable channels $C_{S,R}$ and $C_{R,S}$. The data units submitted by the producer must be presented to the consumer without losses or duplicates and in the same order.

A wide range of *sliding-window* protocols can be used to implement reliable data transfer. These protocols are based on the consecutive numbering of data units and the notion of send and receive windows. The numbers assigned to the data units are called *sequence numbers*. Packets from the sender to the receiver identify the data units they carry by their sequence number. The receiver uses the sequence numbers to put data units in the correct order if some packets are reordered. Sequence numbers are also used in *acknowledgments* (acks, for short) sent by the receiver to the sender to report the receipt or loss of data units.

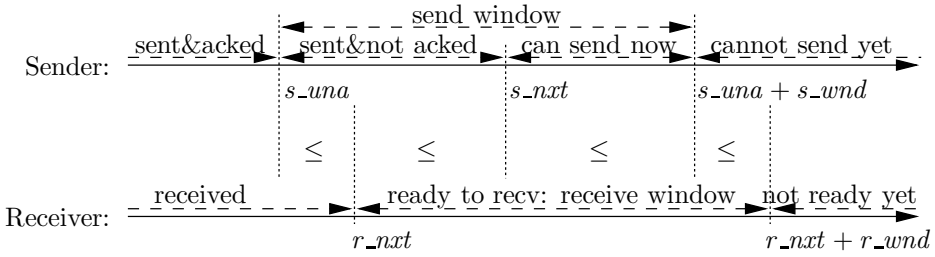


Figure 1.3: *The variables of the sliding-window protocol*

The *receive window* is the range of data units which the receiver is ready to accept at a given moment. The lower- or left edge of the receive window is the lowest sequence number of unreceived data units. The upper- or right edge of the receive window is one larger than the highest sequence number which the sender is allowed to produce at that moment. The receiver informs the sender about the current size of its window in the acks. The *send window* is the window known to the sender in any given moment. Because data packets and acks need time to cross the network, there is a possible phase shift between sender and receiver.

Figure 1.3 shows a snapshot of the state variables of the sliding-window protocol. The variable *s_una* is the left edge of the sender window. All the data units up to but not including *s_una* have been acknowledged. The right edge of the sender window is given by $s\_una + s\_wnd$. The variable *s_nxt* is the sequence number of the first data unit that has not been sent yet. The left edge of the receiver window is *r_nxt*, the right edge of the window is given by $r\_nxt + r\_wnd$.

The above description lists only the important features of sliding-window protocols. Several variations of the sliding-window protocol can be found in practice. Some of these we list here.

Acknowledgments can be either cumulative or selective. The *cumulative ack* (i) acknowledges the receipt of every data unit with a sequence number less than i. Thus a cumulative ack can be generated by writing the value of *r_nxt* to the ack. A selective ack (i, l) acknowledges the receipt of a range of data units of the sequence numbers $[i, \ldots, i + l - 1]$.

The advantage of cumulative acks is their inherent redundancy. If one is lost, the receipt of a subsequent ack will acknowledge the same range of sequence numbers or even more. Therefore this redundancy provides a simple protection against loss. If a selective ack is lost, there is no guarantee that a subsequent ack carries overlapping acknowledgment information.

The disadvantage of cumulative acks is that they are not able to fully describe the state of the sender to the receiver. In particular, cumulative acks cannot inform the sender about the data units received out-of-order. Having full information at the sender about the data received by the receiver is important for efficient error control in high-speed long-delay networks [DJNS93, FB90].

In most protocols, the unit of sequencing is either an octet or a variable sized record which fits into a packet. There is no clear advantage for any of them. Other units, such as bits in Delta-t [FW78], are not used in modern protocols.

A more fundamental modification to the sliding-window protocol has been proposed recently [JBB92]. The idea is to extend the sequence space with *timestamps*. The need for extending the sequence space arose in the context of high-speed networks. Because the sequence numbers in any practical protocol are from a finite space, the too early reuse of sequence numbers may affect correctness. The evident solution [OP91] would be to raise the bound on the sequence numbers, i.e. to use more bits in their representation.

An alternative solution proposed by Jacobson et al [JBB92] was to use timestamps *and* sequence numbers for the identification of data units. The use of timestamps has two advantages:

- It avoids hazards related to the *early reuse of sequence numbers*, because timestamps help detecting old packets.
- The *measurement of round-trip delay* is made easier by the inclusion of timestamps. Maintaining an accurate estimate of the round-trip delay between the sender and the receiver is essential for an effective congestion control [Jac88, Jai92].

Although the 'Protect Against Wrapped Sequence numbers' (PAWS) proposal of Jacobson et al [JBB92] was made specifically for TCP [Pos81b], similar ideas can be found in other protocols. The `syn` and `echo` fields in the Xpress Transport Protocol (XTP) [XTP95] play the same role as the timestamps and echoed timestamps in PAWS. The TP++ protocol [BF93] also uses timestamps, but in a slightly different manner. We will analyze the mechanisms used by these protocol variants in Chapter 3.

1.3.2 Connection management

The role of connection management (CM) is to open and close connections reliably even in the face of losses and duplicates in the network. The requirements below are from a list of requirements for reliable connection management defined by Watson [Wat89]. A more formal description of the connection management service will be given later in Chapter 4. We kept the original numbering of the requirements in [Wat89]:

O1: If no connection exists and the receiver is willing to receive, no duplicate packets from a previously closed connection should cause a new connection to be established and duplicate data to be accepted.

O2: If a connection exists, then no packets from a previously closed connection should be acceptable within the current connection.

C2: A receiving side should not close until it has received all of a sender's possible retransmissions and can respond to them.

C3: A sending side should not close until it has received an acknowledgment of everything what has been sent.

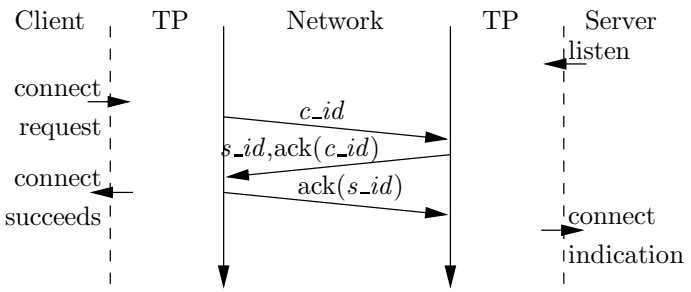


Figure 1.4: *The three-way handshake*

Several protocols are known for solving the problem of reliable connection setup (O1). All of them are based on the idea of assigning *unique identifiers* to open request packets. Such packets, when received, initiate a new connection. The host sending the open request is called the *client*, the host receiving it is the *server*. The question is how the server can decide whether an open request is new or a duplicate.

A well-known connection setup protocol is the three-way handshake [SD78]. This protocol is based on the assumption that the server has possibly no memory of the earlier open requests it has accepted. When the server receives a request with the client's identifier *c_id*, it cannot immediately decide whether the same request has been accepted before. Instead, the server replies with another open request which carries an ack of *c_id* and the server's own unique identifier *s_id*. The client accepts this secondary request if it acknowledges its connection ID. If the server's request was triggered by the reception of an old duplicate primary request, then the *c_id* in the secondary request will not match the current connection ID of the client. The response of the client to a valid secondary request is an ack of *s_id*. Receiving an ack of its connection ID assures the server that the open request was not a duplicate. At this point the server can consider the connection to be open. The sequence of events in a successful three-way handshake is shown in Figure 1.4.

If the server remembers the identifiers of the previously accepted open requests, then the validation of an arriving open request can be done immediately by checking if *c_id*, the unique identifier of the open request, is among the already accepted identifiers. When accepting a request, its identifier is added to the list of old identifiers. The server acknowledges a request when accepted, but attaching its own unique identifier is not necessary in this case. The client opens the connection upon the receipt of an ack of its request.

This sort of connection setup is referred to by many terms in the literature, such as immediate setup, implicit setup and two-way handshake (2WHS). We adopt the last in this thesis. Figure 1.5 shows the sequence of events during the two-way handshake.

A disadvantage of the three-way handshake (3WHS) is the initial delay in setting up a connection. Due to the lack of information, the server has to postpone its decision whether to open the connection when it receives an open request. The delay in opening

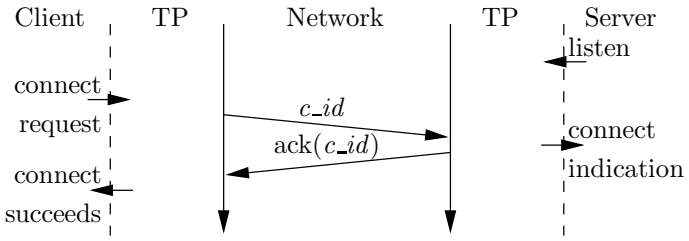
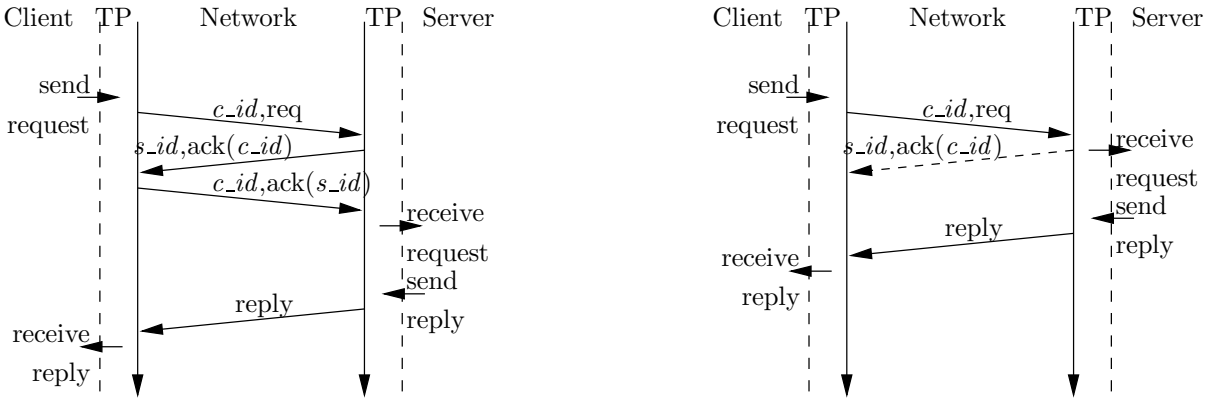


Figure 1.5: *The two-way handshake*



a) Three-way handshake. *b) Two-way handshake.*

Figure 1.6: *Transaction delay as seen by the user in case of different connection setup schemes.*

the connection is one round-trip delay. The negative effect of this delay is increasing as networks become faster. Because the round-trip delay, bounded by the signal propagation time, does not decrease, the amount of data that could potentially be transmitted during the 3WHS is proportional to the network bandwidth.

Looking at it in a different way, eliminating the delay of the 3WHS results in half the latency for transaction-like communications, which is a twofold performance increase in the optimal situation. Transaction-like communications consist of a *short query* sent by the client application which is *followed by a reply* from the server application. As Figure 1.6.a indicates, the minimum latency of such transactions is two times the round-trip delay plus the server processing time. Eliminating the initial delay of the 3WHS would reduce this latency to a single round-trip delay plus the server processing time which is shown in Figure 1.6.b. Transaction-like communications are not uncommon in current networks. The World-Wide Web [BLCL+94] is one application which generates such traffic.

The best known transport protocols using the three-way handshake are TCP [Pos81b, Ste94] and OSI TP4 [Sta90]. Protocols using some form of the two-way handshake are Delta-t [FW78], VMTP [Che88], TP++ [BF93], XTP [XTP95], just to name a few. It is also possible to combine three-way and two-way handshake-based connection setup in a protocol [SL95]. In this case, the table storing the old identifiers acts like a cache. If information about the client is present in the table, then the two-way handshake can be

used; otherwise the protocol falls back to the more time consuming three-way handshake. A protocol based on this scheme is TCP for Transactions (T/TCP) [Bra94, Ste96].

The requirement O2 can be satisfied by sending *c_id* in every data packet during the data transfer phase. The uniqueness of this identifier assures that old packets from previous connections are always detected. In practice, the same identifier can be both the connection identifier and the sequence number of a packet. Such an example is shown in Section 4.4.

The reliable closing of connections requires that C2 and C3 are satisfied. Reliable closing is achieved by sending a close request behind the data units submitted by the user. The receiver can acknowledge the close request only after it has received all the preceding data. The sender is allowed to discard its state information when the close request is acknowledged. The receiver, however, gets no explicit signal to release its state. A timer can be used to keep the state information for a period that is sufficiently long so that the sender can request a retransmission of the closing ack it it is lost. Further thoughts about the closing handshake can be found in Section 4.3.3.

1.3.3 Assumptions about transport protocols

Transport protocols, both connection management and data transfer, use *identifiers* to cope with the possible errors of an unreliable network service. The identifiers can have different forms, such as sequence numbers, timestamps, and connection identifiers. They have an important common feature: all these identifiers are from a possibly very large, but *finite* set.

Although variable-length representations exist which can encode arbitrary large numbers, using such representations in protocols would make implementations overly complicated. Furthermore, unbounded identifiers are not necessary for the correct operation of protocols. Unbounded identifiers are still useful as an abstraction. As we will see later in this thesis, the analysis of protocols is easier to begin at an abstract level assuming unbounded identifiers. The results of this analysis are then helpful to establish the correctness of the real protocol which uses bounded identifiers.

A consequence of the bounded identifier space is that *identifiers must be reused*. The reuse of identifiers affects the protocol mechanisms for connection management and data transfer. In case of connection management, for example, when a new connection identifier is chosen, the protocol must assure that no previous copies exist in the network.

The reuse of identifiers mandates the requirement for the existence of a maximum packet lifetime in the channels (see Section 1.2). Without a known bound on packet delays, identifiers could never be reused without the danger of having an old packet in the network carrying the same identifier.

The number of possible peers for any given host computer is enormous. In case of TCP,

for example, the maximum number of connections originating from a single host is close to 2^{64}. Therefore it is not feasible to keep status information about every previous connection in a host; any connection management protocol must be able to establish a connection without host-specific information in the peers. This assumption may sound trivial, but remember that some of the connection management protocols retain status information about their previous connections. Even those protocols must have a way to handle the situation when no information is known about the peer.

There is another situation when a host must connect to other hosts without access to state information on previous connections. This happens after a so called crash. A crash in our terminology may be the result of a system failure or user action (the computer is switched off). The contents of the memory of a host are lost during a crash. We assume, however, that the hosts are *fail-stop processors*. That is, a host is either operational when it is working according to its specification or it is down when it does not do any processing. Some connection management protocols assume the presence of some safe storage, e.g., a disk. Information saved in the safe storage is not lost during a crash, but the protocols must try to minimize the amount of information saved in the safe storage because of the higher latency to access it.

1.4 Contributions of this thesis

The contribution of this thesis to the design of reliable communications is twofold:

- The formal verification of two recent transport protocols PAWS [JBB92] in Section 3.3 and SCMP [LSW91] in Section 4.2 which have not been verified before.
- Improved understanding of reliable communications through the informal analysis and comparison of other transport protocol mechanisms including the data transfer protocol SNR [NRS90] in Section 3.4 and the connection management protocols T/TCP [Bra92, Bra94] in Section 4.3.5 together with a wider family of cache-based protocols [SL95] in Section 4.3.

1.4.1 Contributions to protocol verification

The formal analysis of two recent protocols is presented which have not been verified before. One is a data transfer protocol called PAWS [JBB92] (Protect Against Wrapped Sequence numbers), the other is a connection setup protocol, SCMP (Synchronized Clock Message Protocol) [LSW91]. The common in these protocols is the innovative use of timestamps.

An assertional verification framework [Sha93] based on a state-transition model and temporal logic is used to verify the protocols. Real-time extensions of this model [Sha94] allow the natural handling of timing aspects of the protocols, such as maximum packet

lifetime, transmission rate, and the rate of opening new connections. Assertional techniques were used successfully before to verify transport protocols [BKKM96, GNS95, MS87, Sha89, SL95, Tel91].

PAWS [JBB92] is an extension to the sliding-window protocol in TCP [Pos81a]. As we mentioned in Section 1.3, the use of timestamps has two advantages: (i) avoids the possibility of the misinterpretation of sequence numbers at high transmission rate, (ii) improves the performance of the protocol over paths which have high bandwidth-delay product.

Although the algorithms in PAWS are intuitively appealing, they are far from trivial. Some informal arguments about the correct behavior of PAWS are presented in the specification [JBB92], but in our view the complexity of such protocols mandates the use of more exact methods.

A specification of the essential mechanisms in PAWS is given in Section 3.3 of this thesis. The correctness of the protocol is then shown by proving safety and progress properties. In fact, our verification reveals that the original protocol may fail to reject duplicates under certain circumstances. A possible fix is proposed. Our verification of PAWS is building on earlier results of Shankar in verifying plain sliding-window protocols [Sha89], i.e. protocols which use only sequence numbers, but no timestamps.

SCMP [LSW91] is a protocol to implement at-most-once message delivery. The novel idea in SCMP is the use of synchronized clocks. Since the major issue when setting up a connection is the rejection of old duplicate requests, the SCMP mechanism can be incorporated into a connection management protocol. Connections can be opened by 2WHS (see Figure 1.5) when SCMP is used.

In our verification of SCMP in Section 4.2, only safety properties are proved. We take into account the finiteness of the timestamps. In particular, we prove the claim that SCMP preserves its correct behavior even if the clocks are out of synchrony [LSW91]. The only requirement for the safety of the protocol is the bounded drift of the clocks.

A formal verification of SCMP is reported by Lampson *et al* in [LLSA93]. They proved both safety and progress properties of SCMP using a model of the protocol which differs from ours. The main difference is that they did not model the finiteness of timestamps and they assumed that the difference of clocks is bounded by a known constant. As it was explained earlier in the introduction, we believe that the boundedness of identifiers is an essential part of protocols, therefore in our verification these issues are treated explicitly. On the other hand, we can not prove progress properties. In fact, it is shown in Section 5.2 that with our assumptions the progress of the protocol cannot be assured.

1.4.2 Contributions to the understanding of protocol mechanisms

Besides the formal verifications discussed above, several other protocols are analyzed informally, both data transfer and connection management protocols. The result of these verifications is never a simple "correct" or "not correct" stamp. The correctness of the protocols depends on the parameters of the underlying network and the protocol itself. A sliding-window protocol, for example, may work correctly with 32-bit sequence numbers in a certain network setup, but may fail with 16-bit sequence numbers. Therefore the verification of these protocols always produces a set of constraints. The protocol operates correctly provided that these constraints are satisfied.

The constraints produced by the verification can be useful in two ways. First, they can be used for planning a communication system. Having a particular transport protocol in mind, one can decide on parameters like the size of the identifiers by simply substituting the parameters of the network in which the protocol has to operate.

Another use of these constraints is to analyze the protocol mechanisms. Obtaining the correctness constraints for different protocols which provide the same service, one can judge the advantages and disadvantages of these protocols. Such an analysis helps to gain further insight into the design of these protocols.

SNR [NRS90] is data transfer protocol designed for high-speed networks. The protocol is based on the idea of periodic state exchange. The specification of SNR presented in [GNS95] assumes that the network can drop and duplicate packets, but it does not reorder them. By comparing SNR to the plain sliding-window protocol and PAWS, we show in Section 3.5 that the assumption of "no duplicates" allows SNR to use its identifier space more efficiently, i.e., achieve higher transmission rate while using identifiers of the same size as the plain sliding-window protocol.

On the other hand, we also show that this assumption can become very expensive in networks which have no strict limits on their maximum packet lifetime because the requirement for avoiding duplicates creates an undesirable coupling between the correctness of the protocol and the retransmission policy that we are allowed to use. Keeping the correctness and performance issues, such as the retransmission policy here, orthogonal eliminates these unwanted side effects. We show in Section 3.4 that the idea of periodic state exchange can be captured by a specification that does not have these limitations.

A wide range of cache based connection management protocols, called SC [SL95] are analyzed in Section 4.3. The common feature of these protocols is that they use the memory for protocol state information as a cache. If the necessary information is present in the cache, then connections are opened by the low latency 2WHS, otherwise the 3WHS must be used. The section is built on the specification and analysis of these protocols by Shankar and Lee in [SL95] and additional work by the author [OHdG96b]. We present modifications that allow to weaken the real-time conditions that must be satisfied for the correctness of the protocols. We also present a technique to reduce the latency of

multi-packet transactions.

We also introduce the notion of *implicit 3WHS* in Section 4.3.4 which becomes the basis of a technique to reduce the memory usage of busy servers. Although this technique does not allow us to shorten the absolute bounds on the necessary caching period, it still allows to shorten the caching period considerably in common situations. In case of busy server servicing many short-lived connections, the savings in memory usage can be substantial.

As a special case of the cache-based protocols, we present a formal model of T/TCP [Bra92, Bra94] in Section 4.3.5. We explain how to model T/TCP using the framework for the wider family of cache based protocols treated earlier in the Chapter. The differences in the caching model of SC and T/TCP are explained. The practical importance of this analysis is that T/TCP is a protocol that is already used in the Internet experimentally [Ste96].

Section 4.4 compares the connection management protocols studied in the thesis. Apart from the conclusions discussed already, another important result is presented there. We show a technique that allows us to eliminate the maximum incarnation lifetime from the condition for the correctness of connection management protocols. This result is very important from the viewpoint of implementing these protocols. The lifetime of a connection incarnation is normally determined by the application that is using the services of the transport protocol, therefore it would not be easy to enforce a bound on the length of connections.

Chapter 2

Verification Framework

Our goal is to talk about transport protocols, to analyze the decisions that must be taken during their design. Formalisms help to specify protocols and their behavior in an unambiguous way. Furthermore they provide methods to verify properties of protocols. This chapter is devoted to the description of the formal verification framework applied in the thesis.

Programs (computing systems) can be partitioned into two classes based on the way they interact with their environment [MP92]. *Transformational programs* calculate a result from their input during a finite computation. Such programs are appropriately characterized by the relation between their input and output. *Reactive programs* do not produce a final result, their task is to maintain an ongoing interaction with their environment. Such programs are best characterized by their ongoing behavior instead of their initial and final states. It is worth mentioning that terminating reactive programs can also be defined which interact with their environment and have distinguished initial and terminal states, but such programs will not be further considered here.

The `grep` program, for example, is a transformational program. Its input is a sequence of lines and a pattern, its output is those lines from the input which match the pattern. On the other hand, a command shell is a typical reactive program. Its input, just as with `grep`, is a sequence of lines, but its expected behavior cannot be captured by the desired output at the termination of the shell. Its most characteristic feature is its interaction with the user: upon reading a line the shell executes the commands in it and reports the result of running these commands back to the user.

Another aspect of systems is *concurrency*. Some programs are executed sequentially in a single process, others are composed of cooperating processes that run in parallel. Programs in the former category are called sequential programs, those in the latter are concurrent programs.

Tel in [Tel91] gave the following classification of concurrent programs:

- In *parallel programs* processes cooperate to achieve a single computational task

faster than it would be possible by sequential processing.

- *Distributed programs* are designed for tasks related to the physical dispersion of the processes and the data operated on. These are thus meaningless in the context of a single sequential process.

Using our terminology, parallel programs are concurrent transformational programs, while distributed programs are concurrent reactive programs. Communication protocols belong to the category of distributed programs.

Assertional techniques characterize behaviors by the sequence of states during the execution of the program. Another way of viewing an execution of a program is as a collection of actions or events [Lam94b]. Event-based formalisms include algebraic approaches like CCS [Mil80] and functional approaches like [Bro93].

The verification framework [Sha93, Sha94] used in this thesis is based on assertional reasoning. Such techniques are widely used for specifying distributed systems [Gou93, Lam94a, MS87, SL95, Tel91]. An overview of assertional verification and specification techniques can be found in [Lam94b].

In this chapter, we proceed as follows. First the verification of traditional sequential programs is discussed. A brief overview of predicate logic and the methods to argue about the behavior of sequential programs is given in Section 2.1. Reasoning about distributed systems involves more complex specifications and properties. Section 2.2 introduces (i) temporal logic as a means to specify properties over sequences of states, (ii) state transition systems as an operational model of concurrent systems, and (iii) methods to prove properties of concurrent systems. In Section 2.3, additional concepts are introduced to tailor the general framework for the purposes of protocol specification and verification. Finally, in Section 2.4 we illustrate the usage of the verification framework through a series of examples.

2.1 Sequential programs

2.1.1 Predicates

Let us consider the set of typed variables $\mathcal{V} = \{u_1, u_2, \ldots, u_n\}$. The *type* of a variable defines the set of values with which it may be associated, called its *domain*. Constants represent specific values over a given domain, such as `true` and `false` over the booleans, $0, 1, \ldots$ over integers.

Expressions are constructed from variables and constants using operators. Expression are also typed according to the domain over which their values range. If x and y are integers, then the expression $x + y$ results an integer, $x < y$ results a boolean value. We will use common mathematical notation for writing expressions without formally specifying them.

Predicates (or state formulas) are boolean expressions which may also contain quantification ($\exists, \forall$) over some variables that appear in the expressions. For example, $x \geq 0 \Rightarrow |x| = x$ and $(\forall i : i > 0 \Rightarrow ix > 0)$ are predicates. The variable i in the second predicate is bound, while x is free.

A *state* $s = \langle x_1 : X_1, x_2 : X_2, \ldots, x_n : X_n \rangle$ assigns a constant to each variable from its domain. A predicate is well-defined in a state if all of its variables are assigned a value in that state. To evaluate a predicate in a state, each variable is replaced by its value and then the resulting constant expression is evaluated according to the definition of the operators and their precedence relations.

If the predicate P evaluates to `true` in state s, the we say that s satisfies P, denoted by $s \models P$. In this way, each predicate specifies the set of those states which satisfy the predicate. Specially, the predicate `true` specifies the set of all states, and the predicate `false` specifies the empty set.

A *tautology* is a predicate which evaluates to `true` in every state. We say that P is valid if it is a tautology. The *De Morgan laws* ($\neg(a \wedge b) = \neg a \vee \neg b$), or the *law of excluded middle* ($a \vee \neg a =$ `true`) are examples of well-known tautologies.

It is impossible, however, to list every valid predicate. A *deductive system* built from axioms and proof rules can be used to prove that a predicate is valid. *Axioms* are predicates which express basic properties of the operators in the language. Their validity is accepted without proof. A comprehensive list of axioms for a predicate calculus can be found in [Gri81, p. 20].

A *proof rule* in the form

$$\frac{P_1, \ldots, P_n}{P}$$

states that the validity of P follows from the validity of $P_1, \ldots, P_n$. The predicates $P_1, \ldots, P_n$ are called the premises, and P is called the conclusion. The *rule of substitution* says that if the predicates p_1 and p_2 are equal (i.e., $p_1 = p_2$ is a tautology), and $E(p)$ is a predicate expressed as a function of one of its variables, then $E(p_1) = E(p_2)$ is also a tautology. With the above notation this can be written as:

$$\frac{p_1 = p_2}{E(p_1) = E(p_2)}$$

2.1.2 Reasoning about the correctness of sequential programs

The correctness of sequential programs can be proven by showing that the program terminates when started from any of the initial states and that the desired relation between the initial and final states is satisfied. The notation of *Hoare-triples* can be used to express properties of sequential programs.

Definition 2.1 (Hoare-triple) *Let P and Q be predicates and S a sequential program. The notation $\{P\}S\{Q\}$ has the following interpretation: If execution of S begun in a state satisfying P, then it is guaranteed to terminate in a state satisfying Q.*

P is called the precondition of S; Q is called the postcondition. For example, the specification $\{x = X \wedge y = Y\}S\{x = Y \wedge y = X\}$ says that program S swaps the values in x and y. Note that a Hoare-triple is a predicate itself which can be either true of false. To show that the program S is correct, we have to prove that the Hoare-triple is valid, i.e., it is true in every state.

Another useful concept for verifying programs is the notion of *weakest preconditions* [Dij76, Gri81]. For any program S and predicate Q, which represents the desired result of executing S, we define a predicate called the weakest precondition of Q with respect to S, denoted by $wp(S, Q)$.

Definition 2.2 (Weakest Precondition) *The predicate $wp(S, Q)$ represents the set of all states such that execution of S begun in any one of them is guaranteed to terminate in a state satisfying Q.*

Using this definition of wp, we can redefine the Hoare-triple $\{P\}S\{Q\}$ as a shorthand notation for $P \Rightarrow wp(S, Q)$.

The predicate transformer wp has some useful properties [Gri81]:

- $wp(S, \texttt{false}) = \texttt{false}$;
- $wp(S, Q) \wedge wp(S, R) = wp(S, Q \wedge R)$;
- if $Q \Rightarrow R$ then $wp(S, Q) \Rightarrow wp(S, R)$;
- $wp(S, Q) \vee wp(S, R) \Rightarrow wp(S, Q \vee R)$.

The above properties assure that wp can be used to define the semantics of the statements in a programming language and the statements will behave as expected. A complete programming language is defined by Dijkstra using wp [Dij76] (see also [Gri81]), but here we give only the definition of some program constructs used later in the protocol specifications.

Assignment statement: $wp(\text{“}x := e\text{”}, Q) = Q[x/e]$, where the notation $P[x/e]$ means that every occurrence of the free variable x in predicate P is replaced by the expression e.

Conditional statement: $wp(\text{“}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\text{”}, Q) = (B \Rightarrow wp(S_1, Q)) \wedge (\neg B \Rightarrow wp(S_2, Q))$.

Sequential composition: $wp(\text{“}S_1;\ S_2\text{”}, Q) = wp(S_1, wp(S_2, Q))$.

The definition of a loop construct is somewhat involved so we do not present it here, but we give a theorem about the weakest precondition of a loop, which should be sufficient for most situations.

Theorem 2.3 (Loop termination) *Consider the loop "**while** B **do** S". Let t be an integer function defined on all states and suppose that function t and predicate P satisfies:*

1. $P \wedge B \Rightarrow wp(S, P)$;
2. $P \wedge B \Rightarrow t > 0$;
3. $P \wedge B \Rightarrow wp(\text{"}t_1 := t; S\text{"}, t < t_1)$, *where t_1 is a fresh identifier.*

Then $P \Rightarrow wp(\text{"}$**while** B **do** $S\text{"}, P \wedge \neg B)$.

Requirement 1 in the theorem assures that P is preserved by an iteration of the loop, 2 requires that t is positive while the loop is executing, and 3 requires that t decreases in every iteration. Since t has a finite value when starting the loop, the number of iterations must be finite.

Weakest preconditions are useful for developing a program along with its correctness proof. However, given an annotated program, a program in which assertions between program statements give properties of the program state in that point of execution, Hoare-triples are sufficient to verify the validity of these assertions.

2.2 Concurrent systems

The behavior of a concurrent system is characterized by a sequence of states. *Temporal logic* [MP92, Sha93] is used to specify such sequences. We want these specifications to be abstract; they must be independent of any particular implementation of the specified behavior. On the other hand, a concurrent program is viewed as a generator of such sequences of states. The specification of a program provides an operational description of the system.

2.2.1 Temporal logic

Just as first order logic can be used to argue about states, temporal logic is a possible tool to argue about sequences of states. A *behavior* $\sigma = \langle s_0, s_1, \ldots \rangle$ is a sequence of states, where each state s_i provides an interpretation for the variables in the set $\mathcal{V} = \{u_1, u_2, \ldots, u_n\}$. A behavior can be either finite or infinite.

Formulas in temporal logic are built up from predicates and temporal operators. We will use the term 'state formula' for predicates when it is important to make the distinction between state formulas evaluated over states and temporal formulas evaluated over behaviors. The meaning of a temporal formula is a Boolean-valued function on behaviors, defined as follows:

- $P(s_0, s_1, \ldots) \equiv P(s_0)$, for any state formula P. Therefore, state formulas over sequences are evaluated in the first state of the sequence.

- $(F \times G)(\sigma) \equiv F(\sigma) \times G(\sigma)$, for any Boolean operator $\times$.
- $\Box F(s_0, \ldots) \equiv (\forall n : n \geq 0 : F(s_n, \ldots))$

  The temporal formula $\Box F$ (*always* F) holds if F is always true—that is, true now and in all future states.
- $\Diamond F(\sigma) \equiv (\neg\Box\neg F)(\sigma)$

  The temporal formula $\Diamond F$ (*eventually* F) holds if F is eventually true—that is, true now or in some future state.
- $(F \rightsquigarrow G)(\sigma) \equiv (\Box(F \Rightarrow \Diamond G))(\sigma)$

  The temporal formula $F \rightsquigarrow G$ (F *leads to* G) asserts that if F ever becomes true, then G will be true then or in some later state.
- $(F \rightarrow G)(\sigma) = (\Box(F \Rightarrow \Box G))(\sigma)$

  The temporal formula $F \rightarrow G$ (F *establishes* G) asserts that if F ever becomes true, then G will be true then and in any later state.

Many more temporal operators, including past operators and quantification over behaviors, are defined in [MP92], but those will not be used in the thesis.

To minimize the number of parentheses, temporal operators have lower precedence than boolean operators. Among temporal operators, the unary operators $(\Box, \Diamond)$ have higher precedence than the binary operators $(\rightsquigarrow, \rightarrow)$. Below are some examples for evaluating temporal formulas, where $\sigma = (\langle x : 1\rangle, \langle x : 2\rangle, \langle x : 3\rangle, \ldots)$:

- $(x < 2)(\sigma) = \texttt{true}$
- $(\Box x < 2)(\sigma) = \texttt{false}$
- $(\Diamond x > 5)(\sigma) = \texttt{true}$
- $(x = N \rightarrow x \geq N)(\sigma) = \texttt{true}$
- $(x = N \rightsquigarrow x \geq 2N)(\sigma) = \texttt{true}$

Note the usage of the *parameter* N in the last two formulas. Parameters provide a convenient way to define a class of formulas. When evaluating such assertions, every parameter is universally quantified, i.e., the last formula is equivalent to $(\forall N : (x = N \rightsquigarrow x \geq 2N)(\sigma)) = \texttt{true}$.

If a formula F evaluates to true over the the behavior σ ($F(\sigma) = \texttt{true}$), then we say that σ satisfies F which is denoted by $\sigma \models F$. Each temporal formula characterizes a set of behaviors, just as a state formula characterizes a set of states. If F evaluates to true over every behavior, then it is said to be a valid formula. A deductive system for proving the validity of temporal formulas is presented in [MP92].

2.2.2 State transition systems

Concurrent programs are specified by a state-transition system and fairness requirements. The state transition system captures the concepts of the system state and the elementary

actions that modify the state. The fairness requirements are additional concepts needed for the realistic modeling of practical systems.

A state transition system $A = \langle \mathcal{V}, \mathcal{E}, \mathcal{I} \rangle$ is defined by:

- $\mathcal{V} = \{u_1, \ldots, u_n\}$ — A finite set of typed state variables.
- $\mathcal{E}$ — A finite set of events.
- $\mathcal{I}$ — An initial condition.

Each event $e \in \mathcal{E}$ represents a single action of the system which is capable to alter the system state. Events are specified by an enabling condition $\mathit{enabled}(e)$ and an action $\mathit{action}(e)$. The enabling condition is a predicate on the state variables, the action is a sequential program that updates the variables in $\mathcal{V}$. Each event defines a *set of state transitions* (s, s'). A pair of states (s, s') is a transition of e if

- s satisfies $\mathit{enabled}(e)$;
- s' is the result of executing $\mathit{action}(e)$ in s.

The execution of events is atomic and it is required that $\mathit{action}(e)$ always terminates when executed in a state satisfying $\mathit{enabled}(e)$. This latter requirement can be rewritten using weakest preconditions as $\mathit{enabled}(e) \Rightarrow \mathit{wp}(\mathit{action}(e), \texttt{true})$.

The initial states are given by the predicate $\mathcal{I}$. The system may start from any state satisfying $\mathcal{I}$ and it is assumed that $\mathcal{I}$ is non-empty.

A state transition system is executed as follows. At start-up, the system is placed in a state s_0 which satisfies $\mathcal{I}$. In any state s_i, the enabling condition of each event is evaluated and one with a true enabling condition, say e_i, is selected non-deterministically and executed. The resulting state is s_{i+1}. If a state s does not satisfy any of the enabling conditions, then s is a terminal state.

Each execution of the above model generates a series of states. A *computation* $\sigma = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ is an alternating sequence of states and events which satisfies the following criteria:

- The first state is initial, i.e., s_0 satisfies $\mathcal{I}$;
- Each pair of consecutive states (s_i, s_{i+1}) in σ is a transition of some $e \in \mathcal{E}$.

The set of computations of system A, $\mathcal{C}(A)$, contains all computations σ that can be generated by the execution of A. $\mathcal{C}(A)$ can be used to characterize the behavior of the system. A finite computation always ends in a state. Any prefix of a computation, which ends in a state, is also a computation.

Each computation σ uniquely defines a behavior σ' which is obtained by omitting the events from σ. A temporal formula F is evaluated over a computation σ by evaluating F over the behavior σ' defined by σ, i.e., $F(\sigma) = F(\sigma')$.

Fairness

Regarding the selection of events for execution, we have only stated so far that one of the enabled events is selected non-deterministically. In many situations, this definition is too vague and allows computations which one would consider unrealistic for a real system. To illustrate this, assume the following system $A = \langle \mathcal{V}, \mathcal{E}, \mathcal{I} \rangle$:

- $\mathcal{V} = \{x : \mathbf{int}, b : \mathbf{bool}\}$;
- $\mathcal{E} = \{e_1, e_2\}$, where $\mathit{enabled}(e_1) = b$, $\mathit{action}(e_1) =$ "$x := x + 1$", $\mathit{enabled}(e_2) = x > 10$, and $\mathit{action}(e_2) =$ "$b :=$ `false`";
- $\mathcal{I} = (x = 0 \wedge b = \texttt{true})$.

The system works as follows: Initially, only e_1 is enabled which increments x by one in each step. When x exceeds 10, e_2 becomes enabled as well. Once e_2 is executed, the system enters a state where no more change of the state variables is possible.

The question is if there is any guarantee that e_2 is eventually executed, i.e., whether every computation of A satisfies the formula $\Diamond\neg b$. If the above events were implemented as concurrent processes in a real system, then one would expect that eventually e_2 is executed because it is highly unlikely that although e_2 is enabled, it is never selected by the process scheduler. The same question can be asked about the system B which is obtained from A by changing $\mathit{enabled}(e_2)$ as $\mathit{enabled}(e_2) = (2|x)$ (i.e., x is even). In this case, e_2 is enabled infinitely often, but not continuously.

Each fairness requirement is a subset of the events $\mathcal{E}$ tagged with "weak fairness" or "strong fairness." Let $E \subset \mathcal{E}$ be a subset of events. The event set E is said to be enabled if any of the events are enabled, i.e., $\mathit{enabled}(E) = (\exists e \in E : \mathit{enabled}(e))$. Similarly, E is disabled if $(\forall e \in E : \neg \mathit{enabled}(e))$. Let $\sigma = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ be an infinite computation. The event set E is enabled infinitely often in σ if E is enabled in an infinite number of states s_i. In temporal logic, this can be written as $\Box\Diamond \mathit{enabled}(E)$. The event set E is said to be executed infinitely often in σ if an infinite number of events belong to E.

Definition 2.4 (Weak Fairness) *A computation σ satisfies weak fairness for E if either*

1. *σ is finite and E is disabled in the last state;*
2. *or σ is infinite and either E occurs infinitely often or is disabled infinitely often in σ.*

Definition 2.5 (Strong Fairness) *A computation σ satisfies strong fairness for E if either*

1. *σ is finite and E is disabled in the last state;*
2. *or σ is infinite and if E is enabled infinitely often in σ, then it occurs infinitely often in σ.*

Informally, weak fairness means that E eventually occurs if it is enabled continuously. Strong fairness means that E eventually occurs if it is enabled infinitely often. In the examples above, weak fairness is required to assure that e_2 eventually happens in the first case. In the second, only strong fairness can assure that e_2 eventually happens.

The set of *allowed computations* $\mathcal{C}_f(A)$, is the subset of $\mathcal{C}(A)$ containing only those computations which satisfy the fairness requirements as well.

2.2.3 Proving system properties

So far, we discussed how to specify sets of behaviors at an abstract level in temporal logic, and at a lower level of abstraction by state transition systems. Now we turn our attention to methods that can be used to prove that a given system satisfies certain desired properties. The set of its allowed computations $\mathcal{C}_f(A)$ characterizes the system A. We say that A satisfies a temporal formula F, if $(\forall \sigma \in \mathcal{C}_f(A) : \sigma \models F)$, i.e., each allowed computation of A satisfies F. The fact that the system A satisfies the formula F is denoted by $A \models F$.

Safety and progress

Properties are classified as *safety* and *progress* properties [Sha93]. Informally, a safety property asserts that nothing bad can happen. Deadlock freedom, for example, is a safety property. On the other hand, a progress property asserts that eventually something good will happen.

Assertional reasoning is concerned with properties P which evaluate to either true or false over every behavior σ.

Definition 2.6 (Safety) *P is a safety property if for any σ, if P holds for σ then it holds for any prefix of σ.*

Definition 2.7 (Progress) *P is a pure progress property if any finite σ can be extended to a sequence that does satisfy P.*

Thus, if a safety property does not hold for a sequence σ, then there is a point in σ where it becomes false. The formula $\Box p$ is a safety property, $\Diamond q$ is a pure progress property. The formula $(\Box p) \wedge (\Diamond q)$ is neither a safety nor a pure progress property. It has been proven, however, that every property can be expressed as the conjunction of a safety property and pure progress property (see e.g., [Lam94b] for details).

Weakest precondition of events

To be able to argue about the effects of executing an event, we extend the definition of weakest preconditions and Hoare-triples for events.

Definition 2.8 (Weakest Precondition of an Event) *For any event e, the predicate $wp(e, Q)$ represents the set of all states such that execution of $action(e)$ begun in any one of them is guaranteed to terminate in a finite amount of time in a state satisfying Q if $enabled(e)$ held in the starting state. That is, $wp(e, Q) \equiv (enabled(e) \Rightarrow wp(action(e), Q))$.*

Predicate P is a *sufficient precondition* of Q with respect to event e, denoted by $\{P\}e\{Q\}$, if for every state s satisfying P, either e is not enabled or the execution of $action(e)$ starting from s terminates in a state satisfying Q. That is, $\{P\}e\{Q\} \equiv \{P \wedge enabled(e)\}action(e)\{Q\}$. Similarly to sequential programs, $\{P\}e\{Q\} = (P \Rightarrow wp(e, Q))$.

Predicate P is a *necessary precondition* of Q with respect to event e if $(\neg P \Rightarrow enabled(e)) \wedge \{\neg P\}e\{\neg Q\}$. That is, for every state s satisfying $\neg P$, e is enabled and execution of $action(e)$ results in a state satisfying $\neg Q$.

If P is both a sufficient and necessary precondition of Q with respect to e, then P is a weakest precondition of Q with respect to e [Sha93].

Proving safety

Now, we introduce some simple rules to prove that a system $A = \langle \mathcal{V}, \mathcal{E}, \mathcal{I} \rangle$ satisfies a safety property P. The rules are presented without proving their soundness here since they are rather intuitive. For a more rigorous treatment, we refer the reader to [Sha93].

Rule (2.1) is the basic rule to prove invariance. If every event preserves the assertion P, then once P becomes true, it remains so for the rest of the computation.

$$\frac{(\forall e \in \mathcal{E} : \{P\}e\{P\})}{A \models P \rightarrow P} \tag{2.1}$$

Combining rule (2.1) and the equivalence $(\Box p) \Rightarrow p$, we get the second rule for proving $\Box P$. This will be used quite often, because most of the protocol properties we are interested in are invariants in this form.

$$\frac{\mathcal{I} \Rightarrow P, A \models P \rightarrow P}{A \models \Box P} \tag{2.2}$$

Suppose, we have already proven that $A \models \Box Q$ for some Q. Rule (2.3) provides a way to incorporate this knowledge into our subsequent proofs. Note that by replacing Q with the constant `true`, rule (2.3) reduces to rule (2.1):

$$\frac{A \models \Box Q, (\forall e \in \mathcal{E} : \{P \wedge Q\} e \{Q \Rightarrow P\})}{A \models P \rightarrow P} \tag{2.3}$$

We list two other rules, similar in nature to the above, which allow the use of proven properties in proving that $P \rightarrow Q$:

$$\frac{A \models \Box(P \Rightarrow Q), A \models P \rightarrow P}{A \models P \rightarrow Q} \tag{2.4}$$

$$\frac{A \models \Box(P \Rightarrow Q), A \models Q \rightarrow Q}{A \models P \rightarrow Q} \tag{2.5}$$

The rules above define sufficient conditions for system A to satisfy some property P. Given A and a desired property P, one usually cannot prove directly from these rules that $A \models P$. Instead, additional properties of the system have to proven first. The goal is to prove enough additional properties of A until we are able to prove the validity of the desired property.

Generating a proof is not a trivial task. It requires insight into the working of the system and a considerable amount of intuition. For proving properties in the form $\Box p$, where p is a state formula, Shankar proposes a heuristic approach in [Sha93]. The approach is based on generating the weakest precondition of a desired invariant p_0 with respect to each event. The procedure generates a series of properties p_i, each being the precondition for some p_j $(j < i)$. The procedure ends when for all e and for all p_i, $wp(e, p_i)$ is implied by the conjunction of the state formulas $p_0, \ldots, p_n$.

Proving progress

Only a limited sort of progress properties will appear in our protocol verifications. Each of them can be written in the form $p \leadsto q$, where p and q are state formulas. Therefore, we will only consider proof rules for such formulas.

The first type of proof rule allows to infer leads-to assertions from the system specification. Let $A = \langle \mathcal{V}, \mathcal{E}, \mathcal{I} \rangle$ be a system, where $E \subseteq \mathcal{E}$ is marked with weak fairness. Then the following rule can be used:

$$\frac{(\forall e \in \mathcal{E} : \{P\} e \{P \vee Q\}), (\forall e \in E : \{P\} e \{Q\}), A \models \Box(P \Rightarrow \mathit{enabled}(E))}{A \models P \leadsto Q} \tag{2.6}$$

The first premise requires that none of the events can falsify P without establishing Q. Furthermore, if any of the events $e \in E$ eventually happen, Q is guaranteed to be established. The last premise assures that E remains enabled until Q becomes true, therefore the weak fairness on E guarantees that an event $e \in E$ eventually happens.

If event set E has strong fairness instead of weak fairness, then the premise $\Box(P \Rightarrow \mathit{enabled}(E))$ can be replaced with the weaker $P \leadsto (Q \vee \mathit{enabled}(E))$:

$$\frac{(\forall e \in \mathcal{E} : \{P\}e\{P \vee Q\}), (\forall e \in E : \{P\}e\{Q\}), A \models P \leadsto (Q \vee \mathit{enabled}(E))}{A \models P \leadsto Q} \tag{2.7}$$

The second type of proof rules are for inferring leads-to assertions from other assertions. These rules are called closure rules, the following examples are from [Sha93]:

$$\frac{A \models \Box(P \Rightarrow Q)}{A \models P \leadsto Q}$$

$$\frac{A \models P \leadsto R, A \models R \leadsto Q}{A \models P \leadsto Q}$$

$$\frac{A \models P \leadsto Q \vee R, A \models R \leadsto Q}{A \models P \leadsto Q}$$

$$\frac{A \models \Box R, A \models P \wedge R \leadsto R \Rightarrow Q}{A \models P \leadsto Q}$$

In some cases, such closure rules have to be applied multiple times, where the exact number is dependent on the actual problem being solved. The *leads-to well-founded closure rule* can be used in such cases, which is a generalization of closure rules based on well-founded structures [Sha93]. A well-founded structure $(Z, >)$ is a partial order $>$ on a nonempty set Z such that there is no indefinite descending chain $z_1 > z_2 > \cdots$ where each $z_i \in Z$.

Let $(Z, >)$ be a well-founded structure, and let $F(w)$ be a state formula with parameter $w \in Z$. Then

$$\frac{A \models P \leadsto Q \vee (\exists x : F(x)), A \models F(w) \leadsto Q \vee (\exists x < w : F(x))}{A \models P \leadsto Q} \tag{2.8}$$

2.3 Modeling transport protocols

In the previous section, a generic model of concurrent systems was given together with methods for specifying and proving properties of such systems. The goal of this section is to refine and specialize the framework for the specification of transport protocols. This consists of (i) introducing new aspects such as auxiliary variables and real-time properties, and (ii) imposing some structural constraints on the generic state-transition system model.

2.3.1 Auxiliary variables

Consider the simple system consisting of a single state variable x, and a single event e which is always enabled and when executed, increments x by 1. Assuming that initially $x = 0$, the system can generate only one behavior $\sigma = \langle x : 0, x : 1, x : 2, \ldots \rangle$.

What can we say about the properties of this system? The formula $x = n \rightarrow x \geq n$ says that x is non-decreasing. This, however, does not fully specify our system, because the behavior $\rho = \langle x : 0, x : 1, x : 1, x : 2, \ldots \rangle$ also satisfies the formula. The temporal operators introduced so far are not sufficient to express that x is strictly increasing.

The problem can be handled in two ways:

- Introduce further temporal operators. In the above case the "next" operator ($\bigcirc$) [MP92] would help.
- Use so-called *auxiliary variables* to express such properties. In our example, one can introduce the integer array $h[0 \ldots \infty]$, which contains only the values -1 initially and each event stores the value of x in the subsequent position of h before incrementing x. The desired property would then become $(i > 0 \wedge h[i] \neq -1) \Rightarrow h[i-1] < h[i]$.

We will use the second method. Some protocol properties are more complex than the above example, and in our view those are easier to understand when formulated with auxiliary variables than a complex temporal formula. A special use of auxiliary variables is the formulation of real-time properties discussed below. The other reason is more subjective: Shankar also proposes the use of auxiliaries in his tutorial on assertional verification of concurrent systems [Sha93] which provides the bulk of the methodology used in this thesis.

The only requirement for auxiliary variables is that they must not influence the behavior of a system in any way. Adding or deleting auxiliary variables from a specification must not have any effect on the set of computations generated by the system. Let $A = \langle \mathcal{V}, \mathcal{E}, \mathcal{I} \rangle$ and $A' = \langle \mathcal{V}', \mathcal{E}', \mathcal{I}' \rangle$ be two systems with the only difference that A' contains some additional auxiliary variables. The set of computations $\mathcal{C}_f(A)$ and $\mathcal{C}_f(A')$ must be identical when removing the references to auxiliary variables in computations $\sigma' \in \mathcal{C}_f(A')$.

Because in our specification only the assignment statement can change the value of a variable, the requirement for auxiliary variables can be enforced by a simple syntactic rule: Auxiliary variables may occur only in assignment statements, and if the right-hand side of the assignment contains auxiliaries then the left-hand side must be an auxiliary as well.

2.3.2 Real-time properties

The state-transition model discussed so far captures the sequence of events and states, but does not say anything about the actual timing. As we have seen in the introduction, during the analysis of transport protocols we cannot avoid the handling of issues related to real time. For example, one of our assumptions about the network service is the existence of an upper limit on packet delays (see Section 1.2). Therefore, the capability to specify such properties must be incorporated into our formal model.

A special class of auxiliary variables can be used to specify real-time properties [Sha94]. The real-valued auxiliary variable τ indicates the time that has elapsed since the initialization of the system. Initially, $\tau = 0$ and it is assumed that at any time during execution τ is equal to an imaginary global real-time clock. τ is not updated by any event, the only valid usage of τ is to assign it to epoch variables which record the (real) time of the occurrence of an event.

An *epoch variable* is a real-valued auxiliary variable that is used exclusively to record the time at which an event occurred. The only valid operation involving an epoch variable t is $t := \tau$. The symbol λ is reserved to denote that an epoch variable contains no valid time value. Epoch variables are usually initialized to λ.

To make these variables useful for the modeling of real-time properties, the *increasing time axiom* (2.9) is introduced. Let t be an arbitrary epoch variable. Every system must satisfy the invariant:

$$\Box(t \neq \lambda \Rightarrow t \leq \tau) \tag{2.9}$$

In words, this invariant means that τ is monotone increasing, i.e., time must not move backwards. We also assume that the execution of an event is instant, it takes no time. That is, if the action of an event contains the sequence of statements $x := \tau; \ldots; y := \tau$, where x and y are epoch variables, then $x = y$ will hold after the event.

During execution a system is alternating between processing of events and "passing time." While processing an event, no changes to τ can be made, thus in the model the execution of an event takes no time. In the "pause" between two events the system is passing time, i.e., an external event that is not shown in the specification increases the value of τ by some non-negative amount. In our model, we do not try to specify further how τ is updated, the only requirement is that time is increasing which is enforced by the increasing time axiom.

With these extensions, real-time properties can be expressed by assertions in the same way as other safety properties. For example, let e_1 and e_2 be two events of system A and t_1 and t_2 be two epoch variables that are updated in e_1 and e_2, respectively. The constraint that e_2 does not occur within T seconds after e_1's occurrence is modeled by the safety property (2.10):

$$A \models \Box(t_1 \neq \lambda \wedge t_2 \neq \lambda \Rightarrow t_2 - t_1 > T) \tag{2.10}$$

The constraint that e_2 must occur within T seconds of e_1's occurrence is modeled by the safety property (2.11):

$$A \models \Box(t_1 \neq \lambda \wedge t_2 = \lambda \Rightarrow \tau - t_1 < T) \tag{2.11}$$

Because the real-time properties of the system are encoded in the state variables, there is no need to introduce special proof rules to reason about real-time properties.

Real-time constraints in the form of (2.10) and (2.11) are treated as axioms, i.e., it is assumed that every computation satisfies them. Therefore, a real-time system A is characterized by the set $\mathcal{C}_{f,t}(A) \subseteq \mathcal{C}_f(A)$ which contains only those computations that satisfy the real-time constraints.

Note that in this way arbitrary real-time constraints can be specified. It is the responsibility of the designer to make sure that the constraints on the model correspond to some properties of the real system which is modeled. If, for example, two events of the state-transition model belong to physically separated entities in the real system, then it may make little sense to specify direct timing relations between the execution of these events. But it cannot be formulated as a general rule. As another example, consider our assumption about the maximum delay of packets in the network. Formally, it can be captured by a time constraint between the events of sending and receiving a packet. Although these events belong to physically separated entities, in this case it is necessary to assume the existence of such a limit.

2.3.3 Structural restrictions

The state-transition model discussed in Section 2.2 places little constraints on the modeled system. This generic model can be used effectively to describe concurrent programs based on different paradigms, such as message passing or shared variables [MP92]. Since our interest is the specification and verification of protocols (distributed systems), we present here a more specific model based on message passing. Each distributed system of the specific model has a corresponding system in the generic state-transition model.

Processes

A distributed system consists of *processes* that are connected by *channels*. The only way of communication between processes is by sending *messages* through the channels. The state variables are owned by the processes, a process cannot access the variables of other processes. The only exception from this rule is the global variable τ representing the real time. It can be read by any process to update its epoch variables. Each process consists of events which correspond to the events in the generic model.

Channels

Channels can be seen as special processes. The state of a channel C is represented by the sequence of messages currently in transit which appears among the state variables of the state-transition system modeling the distributed system. Process events may use the **send** and **recv** primitives for sending and receiving messages, and the **head** primitive to query the status of a channel. The statement $\mathbf{send}(C, m)$ appends the message m to the sequence of channel C, $\mathbf{head}(C)$ returns the first message in C or `nil` is C is empty, and $\mathbf{recv}(C)$ returns and removes the first message from C provided that $\mathbf{head}(C) \neq \texttt{nil}$ holds. The result of $\mathbf{recv}(C)$ is undefined if C is empty.

The channels are *imperfect*, i.e., they can lose, reorder, and duplicate messages in transit. The imperfection of a channel is modeled by events associated with the channel. These events are always enabled when there are messages in transit.

It is assumed that channels are *non-blocking*, meaning that a channel is always ready to accept a new message. Therefore the number of messages that can be in transit is not limited and process events are free to generate messages without any restrictions. On the other hand, any event e that is willing to receive a message from C must satisfy $\mathit{enabled}(e) \Rightarrow \mathbf{head}(C) \neq \texttt{nil}$, otherwise the requirement that $\mathit{enabled}(e) \Rightarrow \mathit{wp}(\mathit{action}(e), \texttt{true})$ would be violated.

The above requirement assures that no event tries to receive a message from an empty channel. In our protocol models, we will also assume that there is always an event ready to receive a message. That is, for every channel C, $\mathbf{head}(C) \neq \texttt{nil}$ implies that at least one event e is enabled and $\mathit{action}(e)$ removes a message from C.

Channel fairness

Let $E_C(m)$ denote the set of events that can receive message m from C. For a set of messages M, let $E_C(M) = \bigcup_{m \in M} E_C(m)$. If the system satisfies $\Box(\mathbf{head}(C) = m \Rightarrow \mathit{enabled}(E_C(m)))$, then we say that the receive events of C are always ready.

For a channel C we assume the following fairness requirement [Sha93]:

Definition 2.9 (Channel fairness) *If the receive events for channel C are always ready, then for every allowed computation σ and for every message set M, if messages from M are sent infinitely often in σ, then messages from M are received infinitely often in σ.*

In practical terms it means that a message is eventually received if it is sent often enough. Channel fairness is somewhat similar to, but not equivalent with strong fairness for the event set $E_C(M)$.

Having defined channel fairness, we can formulate a new proof rule called the *leads-to via message set rule* [Sha93]. Let C be a fair channel whose receive events are always ready and M be a set of messages that can be sent into C. Let the auxiliary variable $count(M)$ denote the number of times M has been sent. Then we can prove that system $A = \langle \mathcal{V}, \mathcal{E}, \mathcal{I} \rangle$ satisfies $P \rightsquigarrow Q$ provided that the following assertions are satisfied:

- $R_1 : (\forall e \in \mathcal{E}\colon \{P\}e\{P \vee Q\})$
- $R_2 : (\forall e \in E_C(M)\colon \{P\}e\{Q\})$
- $R_3 : A \models \Box(P \wedge count(M) = k \rightsquigarrow Q \vee count(M) > k)$

which can be written in the form of a proof rule:

$$\frac{R_1, R_2, R_3}{A \models P \rightsquigarrow Q} \tag{2.12}$$

2.3.4 Specification language

A Pascal-like language is used for the specification of protocols. Since the constructs are simple and the specifications are not intended for automatic parsing, we combine programming language structures with mathematical notation liberally. Hopefully, this makes the specifications easy to understand for humans, because an important goal of these specifications is to describe the protocol mechanisms unambiguously.

A program has the following structure:

program *name*;
    type declarations
    process definitions
    initial condition

The type declarations follow the Pascal syntax. We will use the types:

- **bool**—Boolean
- $[i \ldots j]$—the integers n in the range $i \leq n \leq j$
- **int**—the range $[-\infty \ldots \infty]$
- **nat**—the range $[0 \ldots \infty]$

- **epoch**—real variable from the interval $[0, \infty)$

Arrays and records can be constructed from the simple data types.

The process definitions section contains channel and process definitions. A channel definition has the form:

channel *name*;

Three events are associated with each channel which can delete, duplicate and reorder messages while in transit. These events will not appear in the specifications, but are always assumed to be present. The expression $m \in C$ is true if a message identical to m is in transit in channel C. When real-time properties are relevant, we assume that the channel carries pairs in the form of $\langle m, t \rangle$, where m is a message and t is an epoch variable. The value of t records the time when m was transmitted. Assertion (2.13)

$$\Box(\langle m, t \rangle \in C \Rightarrow \tau - t < L_C) \tag{2.13}$$

states that no packet can stay longer in C than L_C, which is a possible form of the maximum packet-lifetime assumption.

A process is specified with the following syntax:

process *name*;
    type declarations
    variable definitions
    event definitions

Type declarations and variable definitions follow the Pascal syntax and need no extra explanation. An event has the form:

event *name*;
    when *enabling condition* **do**
        action

Events may have optional parameters which is a convenient way to define a collection of events, one for every possible value of the parameter. The enabling condition is a Boolean expression, the action is a sequential program. The action may contain the following well-known Pascal constructs:

- $x := e$—assignment
- **if** *condition* **then** S_1 **else** S_2—conditional statement
- **while** *condition* **do** S—loop construct

To make the descriptions shorter, we will use indentation instead of the keywords 'begin' and 'end' to group statements into blocks. Sometimes it is necessary to refer to a variable of a specific process: $P.x$ denotes variable x of process P.

The initial condition is written in the form:

init *boolean expression*;

2.4 Examples

To conclude the explanation of the verification framework, a few very simple examples are presented which demonstrate many of the concepts we will use later in the protocol analyses.

Example 1

Our first example is a system which contains a single a counter. It is difficult to imagine a system simpler than this.

program *counter*;
    process *count*;
        var c: **int**;
        event *incr*;
            when `true` **do**
                $c := c + 1$
    init $c = 0$;

Here are some safety properties of the system:

- $c = n \rightarrow c \geq n$
- $\Box(c \geq 0)$

The first assertion can be proven directly from the invariant rule (2.1) because the single event *incr* satisfies $\{c = n\}incr\{c \geq n\}$. The second assertion is inferred from the initial condition and the validity of the first assertion using proof rule (2.2).

Assuming weak fairness for the event *incr*, we can obtain the following progress properties:

- $c = n \rightsquigarrow c > n$
- $\Diamond(c \geq n)$

The first assertion is proven from the leads-to via event set rule (2.6), because *incr* is continuously enabled and $\{c = n\}incr\{c > n\}$. To prove the second assertion, it is enough to prove that $c = 0 \rightsquigarrow c \geq n$ because $p \wedge (p \rightsquigarrow q)$ implies $\Diamond q$ and $c = 0$ is the initial condition. To prove $c = 0 \rightsquigarrow c \geq n$, we can use the leads-to well-founded closure rule (2.8) when Z is the set of natural numbers, $F(x) \equiv x \leq n - c$.

Example 2

Let us now extend the above example with real-time constraints, i.e., turn our counter into a clock.

program *clock*;
  process *clk*;
    var *c*: **int**;
      t: array [**nat**] **of epoch**;
    event *tick*;
      when `true` **do**
        $c := c + 1$;
        $t[c] := \tau$
init $c = 0 \wedge t = \langle 0, \lambda, \lambda, \ldots \rangle$;

Each time the clock "ticks," the variable c is incremented and the current time is saved in $t(c)$. The following two assertions express that the clock period (the time between two ticks) is within the (γ, Γ) interval:

$$\Box(t[n] \neq \lambda \wedge t[n+1] \neq \lambda \Rightarrow t[n+1] - t[n] > \gamma) \tag{2.14}$$

$$\Box(t[n] \neq \lambda \wedge t[n+1] = \lambda \Rightarrow \tau - t[n] < \Gamma) \tag{2.15}$$

These assertions cannot be inferred from the specification, they are assumptions about the system. When making such assumptions, it is always necessary to check that the assumptions are realistic, i.e., they are satisfied by implementations of the modeled system. Real-time assumptions of this kind will be used for the verification of the timestamp-based protocols PAWS in Section 3.3 and SCMP in Section 4.2.

Example 3

In this last example, the simple counter of Example 1 is extended to two counters that are roughly synchronized. The synchronization is achieved by message exchanges over unreliable channels. The system consists of two processes P_1 and P_2, connected by two

channels $C_{1,2}$ and $C_{2,1}$. P_1 sends messages to $C_{1,2}$ and receives them from $C_{2,1}$; P_2 does it in the other way around. Since the system is fully symmetric, we specify only P_1. P_2 can be obtained by the systematic exchange of the subscripts 1 and 2.

program *synchronized_counters*;
  channel $C_{1,2}, C_{2,1}$;
  process P_1;
    var c, l: **int**;
    event *incr*;
      when $c \leq l$ **do** $c := c + 1$;
    event *s_sync*;
      when `true` **do** **send**$(C_{1,2}, c)$;
    event *r_sync*;
      when **head**$(C_{2,1}) \neq$ `nil` **do** $l := \max($**recv**$(C_{2,1}), l)$;
  process P_2;
    { identical to P_1 with indices permuted }
  init $P_1.c = 0 \wedge P_1.l = 0 \wedge P_2.c = 0 \wedge P_2.l = 0$;

The working of the system is simple: each process maintains an estimate of the counter of its peer in the variable l. The local counter c can only be incremented when c is not larger than l, thus we expect that the counters remain close to each other. To update the peer's estimate, each process regularly transmits the value of the local counter.

There are two interesting properties of the system to be proven: (i) the counters are indeed synchronized, (ii) the counters keep on counting forever. These properties are captured by the following temporal formulas:

$$(\exists k : \Box |P_1.c - P_2.c| \leq k) \tag{2.16}$$

$$(\forall n : \Diamond P_1.c \geq n) \tag{2.17}$$

Property (2.16), a safety property, says that the difference of the counters is limited by constant k at any moment which means synchronization. Property (2.17), a progress property, says that the counter of P_1 grows beyond any bound. Due to the synchronization property, $P_2.c$ also satisfies this property.

The safety property can be proven through a series of steps. In each step a new assertion is proven until the desired property is obtained. The assertions below are parameterized by i and j, where $\langle i, j \rangle$ is either $\langle 1, 2 \rangle$ or $\langle 2, 1 \rangle$:

- $\mathcal{S}_1 : \Box P_i.c \leq P_i.l + 1$ using the rules (2.1) and (2.2).
- $\mathcal{S}_2 : \Box x \in C_{i,j} \Rightarrow x \leq P_i.c$ using the same rules as above. Note that initially the channels are empty and that none of the channel events (drop, delete, reorder) can falsify $\mathcal{S}_2$.

- $\mathcal{S}_3 : \Box P_i.l \leq P_j.c$ using the rule (2.3) with $\mathcal{S}_2$ as Q.

The combination of $\mathcal{S}_1$ and $\mathcal{S}_3$ implies the desired property (2.16) for $k = 1$.

To prove the desired progress property, it is sufficient to prove the following assertion:

- $\mathcal{P}_1 : P_i.c = n \rightsquigarrow P_i.c > n$. From $\mathcal{P}_1$, we can prove the desired property by using the leads-to well-founded closure rule (2.8).

The assertions below are needed for proving $\mathcal{P}_1$:

- $\mathcal{P}_2 : P_i.l \geq n \rightsquigarrow P_i.c > n$ by applying the leads-to via event set rule (2.6) assuming weak fairness for the event $P_i.incr$.
- $\mathcal{P}_3 : P_i.c \geq n \rightsquigarrow P_j.l \geq n$ by applying the leads-to via message set rule (2.12) assuming channel fairness for $C_{i,j}$ and weak fairness for event $P_i.s\_sync$.

The assertion $\mathcal{P}_1$ can be proven from $\mathcal{P}_2$ and $\mathcal{P}_3$ through the multiple application of closure rules: (i) $P_i.c \geq n \rightsquigarrow P_j.l \geq n$, (ii) $P_j.l \geq n \rightsquigarrow P_j.c \geq n$, (iii) $P_j.c \geq n \rightsquigarrow P_i.l \geq n$, and (iv) $P_i.l \geq n \rightsquigarrow P_i.c > n$.

Chapter 3

Data Transfer Protocols

This chapter discusses protocols for reliable data-transfer over lossy channels. The discussion is based on the formal verification of important variants from the family of sliding-window protocols. Using the results of these verifications, the data transfer part of many current transport protocols is compared and analyzed.

Section 3.1 gives a formal definition of the desired properties of data transfer protocols. Having these desired properties, we can define data transfer protocols as any protocol which satisfies the desired properties.

In Section 3.2, we summarize the verification of a generic sliding-window protocol which uses sequence numbers only. This verification was carried out by Shankar [Sha89]. There are two reasons to include it in the thesis: (1) it establishes the properties of an important class of data-transfer protocols; (2) it demonstrates some basic verification techniques which are used in our verifications later on.

Section 3.3 is devoted to the verification of an extension to the sliding-window protocol, called PAWS [JBB92]. As it is explained in Section 1.3, PAWS extends the sequence-number space of the plain sliding-window protocol with timestamps.

Another data transfer protocol is analyzed in Section 3.4. The protocol, which is called SNR after the name of its inventors [NRS90], is based on the periodic exchange of state information between sender and receiver. The protocol was verified by Gouda *et al* in [GNS95]. Here, the protocol is modified by making it possible to implement the protocol without the costly protocol reset required by the original specification.

Section 3.5 provides an overview of the options available for reliable data-transfer protocols. Based on the results of the formal verifications, these alternatives are compared to each other using measures such as maximum transmission rate, interaction with other protocol functions and complexity of implementation.

```
program PC;
    type dataunit= "the type of data units";
    process P;
        var Source: array [0...∞] of dataunit ∪ empty;
            Acked: array [0...∞] of bool;
        event Produce(d, i);
            when Source[i] = empty do Source[i] := d;
        event Acknowledge(i);
            when ¬Acked[i] do Acked[i] := true;
    process C;
        var Sink: array [0...∞] of dataunit ∪ empty;
        event Consume(d, i);
            when Sink[i] = empty do Sink[i] := d;
    init (∀i : Source[i] = Sink[i] = empty ∧ ¬Acked[i]);
```

Figure 3.1: *Abstract data-transfer service*

3.1 Desired properties

Ideally, the desired properties of a system are expressed by temporal formulas (see Section 2.2). The goal of formulating these properties is to specify the desired behavior of data-transfer protocols without reference to the specific protocol mechanisms. That is, the desired properties specify a *service* and by formal verification it can be proven that a protocol *implements* a service.

The problem of data-transfer can be captured as a producer-consumer system (see Figure 1.2). The physical separation of the producer and the consumer is an integral part of the specification which can be modeled by partitioning the system into two processes representing the producer and the consumer, respectively. The concept of processes, however, cannot be expressed in temporal logic. Therefore, we use a protocol schema *and* temporal formulas to specify a service [Sha91].

Syntactically, a *protocol schema* is a partially specified distributed system. The desired safety and progress properties complete the specification by restricting the set of computations to the desired ones.

The protocol schema specifying the abstract data-transfer service is shown in Figure 3.1. The producer P has two state variables $P.Source$ and $P.Acked$, each of them being an infinite array. $P.Source$ holds either the data units produced or the constant '`empty`' if the corresponding data unit has not been produced yet. $P.Acked[i]$ implies that $P.Source[i]$ has been acknowledged by the consumer. The consumer, C stores the received data units in the array $C.Sink$.

The system has three events. The user can submit a data unit for transmission by invoking $P.Produce$. The data units can be produced in any order, but each unit can be sub-

mitted only once. The delivery of data to the receiving user is modeled by *C.Consume*. Similarly to *P.Produce*, units can be delivered in any order, but no more than once. The event *P.Acknowledge* represents the acknowledging of received units.

The specification in Figure 3.1 provides information about the structure of the data-transfer service, but says nothing about the logical relations among the production, consumption, and acknowledgment of data units. These are captured by the following temporal formulas:

$$C.Sink[n] \neq \texttt{empty} \rightarrow C.Sink[n] = P.Source[n] \tag{3.1}$$

$$P.Acked[n] \rightarrow C.Sink[n] \neq \texttt{empty} \tag{3.2}$$

$$P.Source[n] \neq \texttt{empty} \rightsquigarrow C.Sink[n] \neq \texttt{empty} \tag{3.3}$$

$$C.Sink[n] \neq \texttt{empty} \rightsquigarrow P.Acked[n] \tag{3.4}$$

Only those computations of the protocol schema *PC* represent a computation of the data-transfer service which satisfy these formulas.

The formulas (3.1) and (3.2) are safety properties. The first one specifies that once a data unit is accepted by the consumer, it must be equal to the corresponding unit of the producer then and at any later moment. The second formula requires that a unit is not acknowledged before it is consumed. The formulas (3.3) and (3.4) are the corresponding progress properties. They specify that if a data unit is produced then it is eventually consumed, and that if a data unit is consumed then it is eventually acknowledged.

This specification of the data-transfer service is rather general because it makes no constraints on the order of delivering data units to the consumer. The service can be made more specific by requesting that data units are delivered sequentially. This is how most protocols implement the data-transfer service, although some recent studies investigate the advantages of delivering data not in sequential order [CT90].

The specification of the sequential-delivery data transfer service is shown in Figure 3.2. Note that the position of the data unit being produced, consumed or acknowledged is no more a parameter of the corresponding event. This is now implicitly encoded in the sequence of events. The newly added state variables *P.Nxt*, *C.Nxt*, and *P.Una* hold the index of the next data unit to be produced, consumed, and acknowledged, respectively. The desired safety and progress properties remain the same, i.e., the formulas (3.1–3.4).

3.2 Plain sliding-window protocols

Now we turn our attention to the plain sliding-window protocol using the formal specification and verification of Shankar [Sha89].

```
program PC_2;
    type dataunit= "the type of data units";
    process P;
        var Source: array [0...∞] of dataunit ∪ empty;
            Acked: array [0...∞] of bool;
            Una, Nxt: int;
        event Produce(d);
            when true do Source[Nxt] := d; Nxt := Nxt + 1;
        event Acknowledge;
            when true do Acked[Una] := true; Una := Una + 1;
    process C;
        var Sink: array [0...∞] of dataunit ∪ empty;
            Nxt: int;
        event Consume(d);
            when true do Sink[Nxt] := d; Nxt := Nxt + 1;
    init (∀i : Source[i] = Sink[i] = empty ∧ ¬Acked[i])∧ P.Una = P.Nxt = C.Nxt = 0;
```

Figure 3.2: *Data-transfer service with in-order delivery*

3.2.1 Protocol specification

The specification of the plain sliding-window protocol can be found in the Figures 3.3, 3.4, and 3.5.

State variables

The program SW consists of two processes, the sender S and the receiver R, and two channels, $C_{S,R}$ from S to R and $C_{R,S}$ from R to S. The specification starts with the definition of several types. The type '*packtype*' enumerates the different packet types. The packets are stored in a variable record type which is called '*packet*.' The value of the selector *Type* determines the members of the record which are the appropriate header fields of the corresponding packet.

The state variables are a superset of the state variables in the protocol schema *PC_2*. The sender S has a new variable *Wnd* to store the latest known size of the receiver window. The arrays of epoch variables, t_S, t_A, and t_L are needed to handle the real-time properties: $t_S[i]$ records the time when unit i was submitted by the user, $t_L[i]$ records the time when unit i was last sent, and $t_A[i]$ records the time when the acknowledgment of unit i was accepted by S.

Similarly to the sender, the receiver R has a variable *Wnd* to store the current size of the receive window. The receive window gives the number of data units that are acceptable starting from *Nxt*. The variable t_R is an array of epoch variables, where $t_R[i]$ records the

program *SW*;
type *dataunit* = "the type of data units";
packtype = (*DATA*, *ACK*, *SACK*);
packet = **record of**
case *Type*: *packtype* **of**
DATA:
Seq: **int**;
Len: [0 . . . *RW*];
Data: **array** [0 . . . *RW*] **of** *dataunit*;
ACK:
Ack: **int**;
Wnd: [0 . . . *RW*];
SACK:
Ack: **int**;
Len: [0 . . . *RW*];

channel $C_{S,R}$, $C_{R,S}$;

process *S*;
var *Source*: **array** [0 . . . ∞] **of** *dataunit* ∪ `empty`;
Acked: **array** [0 . . . ∞] **of bool**;
t_S, t_A, t_L: **array** [0 . . . ∞] **of epoch**;
Una, *Nxt*: **int**;
Wnd: [0 . . . *RW*];
See Figure 3.4 for the events of process *S*.

process *R*;
var *Sink*: **array** [0 . . . ∞] **of** *dataunit* ∪ `empty`;
t_R: **array** [0 . . . ∞] **of epoch**;
Nxt: **int**;
Wnd: [0 . . . *RW*];
See Figure 3.5 for the events of process *R*.

init $(\forall i : S.Source[i] = R.Sink[i] = \texttt{empty} \wedge \neg S.Acked[i]) \wedge$
$(\forall i : S.t_S[i] = S.t_A[i] = S.t_L[i] = R.t_R[i] = \lambda) \wedge$
$S.Una = S.Nxt = S.Wnd = R.Nxt = R.Wnd = 0$;

Figure 3.3: *Plain sliding-window protocol: main part*

```
event Accept(d);
    when Nxt < Una + Wnd do
        Source[Nxt] := d; t_S[Nxt] := τ; Nxt := Nxt + 1;

event SendD;
    var D: packet;
    when Una < Nxt do
        Select i, l such that Una ≤ i < i + l ≤ Nxt;
        D.Type := DATA; D.Seq := i; D.Len := l;
        D.Data := Source[i ... i + l − 1];
        send(C_{S,R}, D); t_L[i ... i + l − 1] := τ;

event RecACK;
    var A: packet;
    when head(C_{R,S}) = A ∧ A.Type = ACK do
        if Una < A.Ack ≤ Nxt then
            Acked[Una ... A.Ack − 1] := true;
            t_A[Una ... A.Ack − 1] := τ;
            Una := A.Ack; Wnd := A.Wnd;
        else if Una = A.Ack then
            Wnd := max(Wnd, A.Wnd);
event RecSACK;
    var A: packet;
    when head(C_{R,S}) = A ∧ A.Type = SACK do
        if Una < A.Ack < A.Ack + A.Len ≤ Nxt then
            Acked[A.Ack ... A.Ack + A.Len − 1] := true;
```

Figure 3.4: *Plain sliding-window protocol: sender events*

time when unit i was received by R.

The initial condition is self explanatory.

Sender events

The user is allowed to generate a new data word by activating the *Accept* event of process S whenever the window is not closed. The *Accept* event records the new data word in $Source[Nxt]$, the current time in $t_S[Nxt]$ and then increments Nxt.

SendD can be activated if there is unacknowledged data. The protocol allows any retransmission strategy, i.e. any word in the send window $[Una \ldots Nxt - 1]$ may be sent. A suitable range of data is selected first from the send window, then a data packet is constructed and sent. After sending the packet, the current time is recorded in the range

```
event ExpandWindow;
    when Wnd < RW do Wnd := Wnd + 1;

event RecD;
    var D: packet;
    when head(C_{S,R}) = D ∧ D.Type = DATA do
        if Nxt < D.Seq + D.Len ≤ Nxt + Wnd then
            j := max(Nxt, D.Seq);
            Sink[j ... D.Seq + D.Len − 1] := D.Data;
            while R.Wnd > 0 ∧ Sink[Nxt] ≠ empty do
                t_R[Nxt] := Nxt + 1; Wnd := Wnd − 1;

event SendACK;
    var A: packet;
    when true do
        A.Type := ACK; A.Ack := Nxt; A.Wnd := Wnd;
        send(C_{R,S}, A);

event SendSACK;
    var A: packet;
    when There exists i, l such that Nxt < i < i + l ≤ Nxt + Wnd,
            Sink[i − 1] = empty and Sink[i ... i + l − 1] ≠ empty do
        A.Type := SACK; A.Ack := i; A.Len := l;
        send(C_{R,S}, A);
```

Figure 3.5: *Plain sliding-window protocol: receiver events*

of epoch variables $t_L[i \ldots i+l-1]$.

The *RecACK* and *RecSACK* events describe the validation of ack packets. A cumulative ack is acceptable if it acknowledges data that has been sent, i.e., $A.Ack$ is inside the $[S.Una+1 \ldots S.Nxt]$ range. In case an ack does not advance $S.Una$, it may still update the send window $S.Wnd$. Note that $S.Wnd$ can only decrease if $S.Una$ is increased by at least the same amount, therefore $S.Una + S.Wnd$ is monotone increasing. A selective ack may update the acknowledgment status of some data units within the send window, but it never updates the left window edge, $S.Una$.

Receiver events

Activating the *ExpandWindow* event indicates that the user is willing to accept more data. The enabling condition assures that the window does not grow beyond the maximum window size, RW.

Data packets are processed by *RecD*. A packet is acceptable if the data it carries falls within the receive window. The array *R.Sink* is updated if a valid data packet is received and then *R.Nxt* is advanced.

The events *SendACK* and *SendSACK* generate acks of the appropriate type. Generation of cumulative acks is always enabled, but selective acks can only be generated if there is out-of-order data at the receiver.

3.2.2 Verification steps

The goal of the verification is to prove that

- the specification is correct, i.e., it satisfies the desired properties;
- the protocol can be implemented using a finite (modulo-N) representation of the identifiers without compromising its correctness.

Therefore, in the first step of the verification the correctness of the specification is proven assuming unbounded identifiers. For the verification of the modulo-N protocol, the following steps are proposed in [Sha89]:

- Define sufficient conditions under which the unbounded identifiers can be replaced by modulo-N identifiers without affecting the protocol's behavior. The conditions are called correct interpretation (CI) conditions.
- Prove that the unbounded protocol satisfies the CI conditions.

The modulo-N version of the protocol is obtained by replacing the unbounded non-auxiliary variables by modulo-N variables. In case of the sliding-window protocol, this affects the *S.Una*, *S.Nxt*, and *R.Nxt* and the *D.Seq*, *A.Ack* header fields in the packets. The auxiliary variables may remain unbounded, because they are not implemented and by definition their value does not influence the behavior of the system.

The arithmetic operations are also replaced by their modulo-N counterparts. In case of our protocols, it means only addition and substraction. The only problematic issue is how to evaluate comparisons on the modulo-N numbers without changing the result of the unbounded comparison. To handle this, we introduce further desired properties, the so-called correct interpretation (CI) conditions [Sha89], which assure that the modulo-N comparisons provide the same result as the unbounded ones. To obtain the CI conditions, the following result can be used [Sha89]:

> Let a and b be unbounded integers, and let a' and b' be their modulo-N counterparts, respectively. The test $a > b$ of unbounded numbers can be replaced by an equivalent test on their modulo-N counterparts if there exists a constant K which is less than N and
>
> $$b + K \geq a \geq b + K - N + 1$$

The equivalent modulo-N test is $K \geq a' -_N b' > 0$, where $-_N$ means modulo-N subtraction.

In order to be able to implement the comparisons of the modulo-N protocol, the constant K must exist. Therefore a CI condition must be added to the set of desired properties for each comparison in the form of $a_i > b_i$ of the unbounded protocol

$$(\exists K_i : K_i < N : \Box(b_i + K_i \geq a_i \geq b_i + K_i - N + 1)) \tag{3.5}$$

The subscripts are used because for each comparison in the protocol a different constant may be needed. The protocol can be implemented using modulo-N identifiers if the unbounded specification satisfies the CI conditions.

3.2.3 Safety and progress of the unbounded protocol

As it was discussed in Section 2.2, the proof that the protocol satisfies the desired properties consists of the generation of several intermediate assertions which help establish the validity of the desired properties. The most important intermediate assertions are listed below from Shankar's proof. The rest of the verification, which is not detailed here, consists of substituting these assertions into the various proof rules. This step can be done mechanically. More details can be found in the reference [Sha89].

Safety

Assertion (3.6) formulates basic properties of the sliding-window protocol:

$$\Box(S.Una \leq R.Nxt \leq S.Nxt \leq S.Una + S.Wnd \leq R.Nxt + R.Wnd) \tag{3.6}$$

Because of the in-sequence delivery $\Box(0 \leq n < S.Una \Rightarrow S.Acked[n])$ holds for $S.Una$ and similarly $\Box(0 \leq n \leq R.Nxt \Rightarrow R.Sink[n] \neq \texttt{empty})$ and $\Box(0 \leq n \leq S.Nxt \Rightarrow S.Source[n] \neq \texttt{empty})$ hold for $R.Nxt$ and $S.Nxt$, respectively. The first half of the assertion (3.6) $S.Una \leq R.Nxt \leq S.Nxt$ is a precondition of the desired properties and can be considered as a consequence of in-sequence delivery.

The second half of the assertion $S.Nxt \leq S.Una + S.Wnd \leq R.Nxt + R.Wnd$ specifies correct flow control. It assures that only requested data units are sent. Note that the relations enforced by (3.6) between the state variables are also shown in Figure 1.3.

The following three assertions define the meaning of packet headers:

$$\Box(D \in C_{S,R} \wedge D.Type = DATA \Rightarrow$$

$$\Rightarrow \quad D.Data = S.Source[D.Seq \ldots D.Seq + D.Len - 1] \neq \texttt{empty}) \tag{3.7}$$

$$\Box(CA \in C_{R,S} \wedge CA.Type = ACK \Rightarrow$$
$$\Rightarrow \quad R.Sink[0 \ldots CA.Ack - 1] \neq \texttt{empty}) \tag{3.8}$$

$$\Box(SA \in C_{R,S} \wedge SA.Type = SACK \Rightarrow$$
$$\Rightarrow \quad R.Sink[SA.Ack \ldots SA.Ack + SA.Len - 1] \neq \texttt{empty}) \tag{3.9}$$

In the above assertions, D, CA and SA stand for data, cumulative ack and selective ack packets, respectively. From now on, we will stick to this notation which allows us to omit the clause for checking the packet type from assertions.

The last three assertions are preconditions of the so-called non-interference property of cumulative and selective acks [Sha89], which states that $S.Una$ is never updated so that it points to an already acknowledged data word:

$$\Box(CA, SA \in C_{R,S} \Rightarrow$$
$$\Rightarrow \quad CA.Ack \notin [SA.Ack \ldots SA.Ack + SA.Len - 1]) \tag{3.10}$$

$$\Box(SA \in C_{R,S} \Rightarrow$$
$$\Rightarrow \quad S.Una \notin [SA.Ack \ldots SA.Ack + SA.Len - 1]) \tag{3.11}$$

$$\Box(CA \in C_{R,S} \wedge CA.Ack \geq S.Una \Rightarrow$$
$$\Rightarrow \quad \neg S.Acked[CA.Ack]) \tag{3.12}$$

From these assertions and some less interesting ones, the validity of (3.1) and (3.2) can be proven using the proof rules from Section 2.2. There is only one aspect of the proof which we want to mention. Remember that the channels $C_{S,R}$ and $C_{R,S}$ have their own events which must be taken into account during the proof. Obviously, the channel events can affect only those assertions which refer in some way to the contents of the channels. In the above assertions, all such references are made in the form of

$$\Box(P \in C \wedge s_1 \Rightarrow s_2)$$

where P is a packet, C a channel, and $s_{1,2}$ are state formulas. Notice that none of the channel events (*Drop*, *Reorder*, and *Duplicate*) can falsify these assertions.

Progress

The desired progress properties of the generic data-transfer protocol schema are given by assertions (3.3) and (3.4) in Section 3.1. Here we reformulate these assertions without explicit reference to the history variables $S.Source$, $R.Sink$, and $S.Acked$ in the assertions (3.13) and (3.14). In these assertions, we exploit the fact that the sliding-window protocol transfers and acknowledges data in sequential order.

$$S.Nxt > R.Nxt = n \rightsquigarrow R.Nxt > n \tag{3.13}$$

$$R.Nxt > S.Una = n \rightsquigarrow S.Una > n \tag{3.14}$$

$$R.Nxt = n \wedge R.Wnd > 0 \rightsquigarrow S.Una + S.Wnd > n \tag{3.15}$$

Furthermore, another desired property (3.15) is introduced. This property specifies that the sender is eventually informed when the receiver has more space in its window. Without (3.15) deadlock could occur in the case when the receiver has a non-zero window but the sender is never notified about it because of a lost acknowledgment, for example.

The validity of assertions (3.13–3.15) can only be proven assuming that the entities satisfy one out of three liveness assumptions. In formulating these assumptions, the notation $(\#P : e)$ will be used to denote the number of packets P sent so far which satisfy the state-formula e.

In the first liveness assumption, the sender and receiver both handle retransmissions:

$$\begin{aligned} &S.Nxt > S.Una = n \wedge (\#D : n \in [D.Seq \ldots D.Seq + D.Len - 1]) = k \rightsquigarrow \\ &\quad \rightsquigarrow \quad S.Una > n \vee (\#D : n \in [D.Seq \ldots D.Seq + D.Len - 1]) > k \qquad (3.16) \\ &R.Nxt = n \wedge R.Wnd > 0 \wedge (\#CA : CA.Ack = n \wedge CA.Wnd > 0) = k \rightsquigarrow \\ &\quad \rightsquigarrow \quad R.Nxt > n \vee (\#CA : CA.Ack = n \wedge CA.Wnd > 0) > k \qquad (3.17) \end{aligned}$$

(3.16) is a requirement that S eventually (re)transmit data messages which carry the unit next to be acknowledged. (3.17) requires R to send acks eventually as long as its window is non-zero.

In the second assumption, the sender handles all retransmissions and the receiver is only required to eventually respond to received messages. In addition to (3.16), we have the following requirements:

$$\begin{aligned} &S.Wnd = 0 \wedge (\#D : D.Len = 0) = k \rightsquigarrow \\ &\quad \rightsquigarrow \quad S.Wnd > 0 \vee (\#D : D.Len = 0) > k \qquad (3.18) \\ &R.MsgRcvd \wedge R.Nxt = n \wedge (\#CA : CA.Ack = n) = k \rightsquigarrow \\ &\quad \rightsquigarrow \quad R.Nxt > n \vee (\#CA : CA.Ack = n) > k \qquad (3.19) \end{aligned}$$

(3.18) requires S to probe the receiver with data messages of zero length[1] whenever its window is zero. This is needed to assure that a lost window update does not cause deadlock. Let *R.MsgRcvd* be a boolean state variable of the receiver which is true if and only if no ack has been sent since the reception of the last data message. (3.19) instructs R to send an ack eventually when *R.MsgRcvd* is true.

[1]For readability, the generation of window probes and the state variables introduced below are not included in the specification in the Figures 3.3–3.5.

In the third liveness assumption, the receiver handles all retransmissions and the sender is only expected to eventually respond to a received ack message. This assumption consists of (3.17) and the following:

$$\begin{aligned} &S.AckRcvd \wedge S.Nxt > S.Una = n \wedge \\ &\wedge (\#D : n \in [D.Seq \ldots D.Seq + D.Len - 1]) = k \leadsto \\ &\quad \leadsto \quad S.Una > n \vee (\#D : n \in [D.Seq \ldots D.Seq + D.Len - 1]) > k \end{aligned} \tag{3.20}$$

The state variable $S.AckRcvd$ is true if and only if the lowest data word to be acknowledged has not been sent yet since the reception of the last ack message.

The progress verification consists of actually three proofs, one for each set of liveness assumptions. All the three proofs are based on the repeated application of the 'leads-to via message set rule' (2.12). Further details of these proofs can be found in the reference [Sha89].

3.2.4 Correct interpretation conditions

Two CI conditions are necessary to prove the correctness of the modulo-N sliding-window protocol. One corresponds to the comparison in the $R.RecD$ event, the other corresponds to the comparisons in the $S.RecACK$ and $S.RecSACK$ events. These latter two comparisons are covered by a single CI condition. The CI conditions are

$$\Box(D \in C_{S,R} \Rightarrow D.Seq \geq R.Nxt + RW - N + 1) \tag{3.21}$$

$$\Box(A \in C_{R,S} \Rightarrow A.Ack \geq S.Nxt - N + 1) \tag{3.22}$$

The CI condition (3.21) can be obtained using the following argument:

$$R.Nxt < D.Seq + D.Len \leq R.Nxt + R.Wnd$$

is the comparison that has to be evaluated on modulo-N numbers. We expect the upper bound to be always true during the correct operation of the protocol because of the invariant (3.6) proven earlier. For the CI condition which belongs to the "$R.Nxt < D.Seq + D.Len$" comparison, let $K = R.Nxt + R.Wnd$ which is a known upper bound of $D.Seq + D.Len$. Substituting K into the formula (3.5), we get

$$\Box(R.Nxt + R.Wnd + N - 1 \leq D.Seq + D.Len \leq R.Nxt + R.Wnd) \tag{3.23}$$

The CI formula (3.21) is stronger than the lower bound in (3.23) because $\Box(R.Wnd \leq RW)$ and $\Box(D.Len > 0)$. If we can prove the validity of the stronger property (3.21), then the CI condition (3.23 above is also valid.

The CI condition for the *S.RecACK* and *S.RecSACK* events can be obtained using similar arguments. In this case, we use the property that $\Box(A.Ack \leq S.Nxt)$.

The validity of the CI conditions has been proven by Shankar for two cases. In one case, the protocol operates over so-called *transport channels* which can drop, reorder or duplicate packets. In the other case, the protocol operates over so-called *data-link channels* which can only drop or duplicate packets, but not reorder them.

Over transport channels, the CI conditions are valid if

$$N \geq 2RW + L \cdot B \tag{3.24}$$

where N is the size of the sequence space, RW is the maximum window size, L is the maximum packet lifetime, and B is the maximum transmission rate measured in data units per second. The proof that (3.24) implies the validity of the CI conditions uses two real-time assumptions about the protocol. The maximum packet lifetime assumption was discussed in Section 2.3. The other assumption is the existence of a maximum transmission rate B which can be captured by the assertion

$$\Box(S.t_L[i] \neq \lambda \wedge S.t_L[i+1] \neq \lambda \Rightarrow S.t_L[i+1] - S.t_L[i] > 1/B) \tag{3.25}$$

There is no need for real-time assumptions to prove the validity of the CI conditions for data-link channels. In this case, the CI conditions are valid if

$$N \geq 2RW \tag{3.26}$$

This completes our summary of the results of Shankar in [Sha89]. We now turn our attention to the verification of another data-transfer protocol which extends the sliding-window protocol with timestamps.

3.3 Timestamp-based extensions

The increasing network bandwidth affects the sliding-window protocol in at least two ways. On one hand, the higher bandwidth influences the *correctness* of the protocol through the condition (3.24). In case of TCP [Pos81a], for example, L is 120 sec, RW is 2^{16}. Therefore, the maximum allowable transmission rate is at around 300 Mbit/s. It shows that there is no reason to worry about the correctness of current data transfers over the Internet, but it may not be far ahead in the future that transmissions at such a speed will be feasible.

There are a number of ways to secure the sliding-window protocol in high-speed networks:

1. decrease L, i.e., enforce tighter control over packet lifetimes;
2. increase N, i.e., use more bits for the representation of sequence numbers;
3. use some other methods to protect against the misinterpretation of wrapped sequence numbers.

In many cases, tight control of L may not be preferred because it requires additional functionality inside the network. In the Internet community, for example, the trend is to strip the amount of processing that must be performed by routers on every packet to the absolute minimum in order to achieve high-speed transmission at low cost. A typical example of this trend is the simplified lifetime enforcement mechanism in IPv6 [DH95]. The TTL (time to live) field of IPv4 [Pos81a] is replaced by a simple hop count in IPv6.

The second option, increasing N is a viable solution to the problem. It will be compared to other possibilities in Section 3.5.

In the PAWS (Protect Against Wrapped Sequence numbers) proposal [JBB92], the authors follow the third possibility. They add timestamps to the sliding-window protocol which are used to detect old duplicates and thus to prevent the misinterpretation of sequence numbers. Their reason for using timestamps was to address another effect of the high transmission speed on the sliding-window protocol.

The increased bandwidth also affects the *performance* of the sliding-window protocol. Accurate measurement of the round-trip time (RTT) is essential for an efficient retransmission strategy [Jac88]. Traditionally, TCP implementations measure the RTT by timing the difference between the sending of a packet and the reception of its acknowledgment [WS95]. Because TCP uses cumulative acks only, at most one RTT measurement per a send window can be made reliably [Zha86]. In order to use the available bandwidth, the size of the window must be at least as big as the *bandwidth-delay product*. Therefore, in high-speed networks, especially on long-delay paths, the optimal size of the window can be rather large which can adversely influence the accuracy of the RTT measurements. That is why the use of timestamps is proposed in [JBB92]. This mechanism, called PAWS (Protect Against Wrapped Sequence numbers), will be formally specified and analyzed below. The results of our analysis were reported earlier in [OHdG95a, Olá95].

3.3.1 Operation of PAWS

The sliding-window protocol specified in the previous section is simplex because it can transmit data in only one direction. TCP is a duplex data-transfer protocol: each TCP entity behaves as the composition of a sender and a receiver of the sliding-window protocol. Accordingly, the TCP packets are equivalent to the composition of a data packet in one direction and a cumulative ack packet in the other direction. The sequence numbers used for the forward and backward direction are largely independent. The PAWS mechanism is defined for this duplex and symmetric data-transfer protocol.

In addition to the state variables used by the plain sliding-window protocol, each protocol entity in PAWS maintains a monotonic counter or in other words a clock *Clk*. When a packet is transmitted, the current value of the clock is put into the *Ts* field of the packet header. The state of each entity also includes the most recent timestamp *TsRec* from the incoming data stream. *TsRec* is used to validate received packets. Only those packets are accepted which have a *Ts* field not lower than *TsRec*. The packets which pass this check are further examined according to the validation procedure of the sliding-window protocol. Apart from the timestamp *Ts*, each packet carries the so-called echoed timestamp *TsEcho* in its header which gets its value from *TsRec*. *TsEcho* is used by the round-trip delay estimation algorithm [JBB92, Ste94].

A crucial part of the protocol is the method for updating the value of *TsRec*. When specifying this method, one has to take into account that *TsEcho* serves two purposes: round-trip time measurement and validation of sequence numbers. For RTT measurements, it must be taken into account that TCP implementations often delay acks in the hope of reducing the number of packets. This delay should be included into the measured RTT to avoid spurious retransmissions. For the validation of sequence numbers, *TsRec* should be recent enough to reject old duplicates. On the other hand, we should avoid dropping a packet if two packets are reordered in the network; therefore we cannot simply store the timestamp of the most recently received packet in *TsRec*.

The following algorithm was proposed [JBB92, Bra93] that complies with the above requirements. Another state variable *LAck* is maintained which contains the acknowledgment field *Ack* from the last packet sent. *TsRec* is set to *Ts* from the packet header whenever $Seq \leq LAck$ and $Ts > TsRec$ hold, where *Seq* is the sequence number of the first data word in the packet.

Our specification differs from the original PAWS mechanism [JBB92] in one point. For the validation of acks, we use the echoed timestamp present in all packets. Similarly to the recent timestamp for data sequence numbers, each entity maintains the most recent echoed timestamp *TsEchoRec*. The ack part of an incoming packet is accepted only if the echoed timestamp is not lower than *TsEchoRec*.

On the contrary, the original PAWS mechanism validates acks by using the same timestamp check as for data, i.e. if $Ts \geq TsRec$ then the ack is accepted if it is inside the send window. Because *TsRec* is updated only when the data sequence numbers advance, it is possible in theory that the acknowledgment sequence numbers wrap around before *TsRec* is updated. In this case, *TsRec* is not sufficient to distinguish old acks from new ones.

Such a situation is shown in Figure 3.6. x and y are the sequence numbers in the two directions, l is the packet length, x' denotes a wrapped sequence number, i.e. $x' \equiv x \pmod{N}$. Each packet is represented by the $(Ts, Seq, TsEcho, Ack)$ tuple. Each entity is sending pairs of packets in turns. While all packets are delivered from A to B, the first of the two packets from B to A are always lost. This results in A receiving out-of-order packets exclusively, while B keeps on retransmitting the same pair of packets. In this

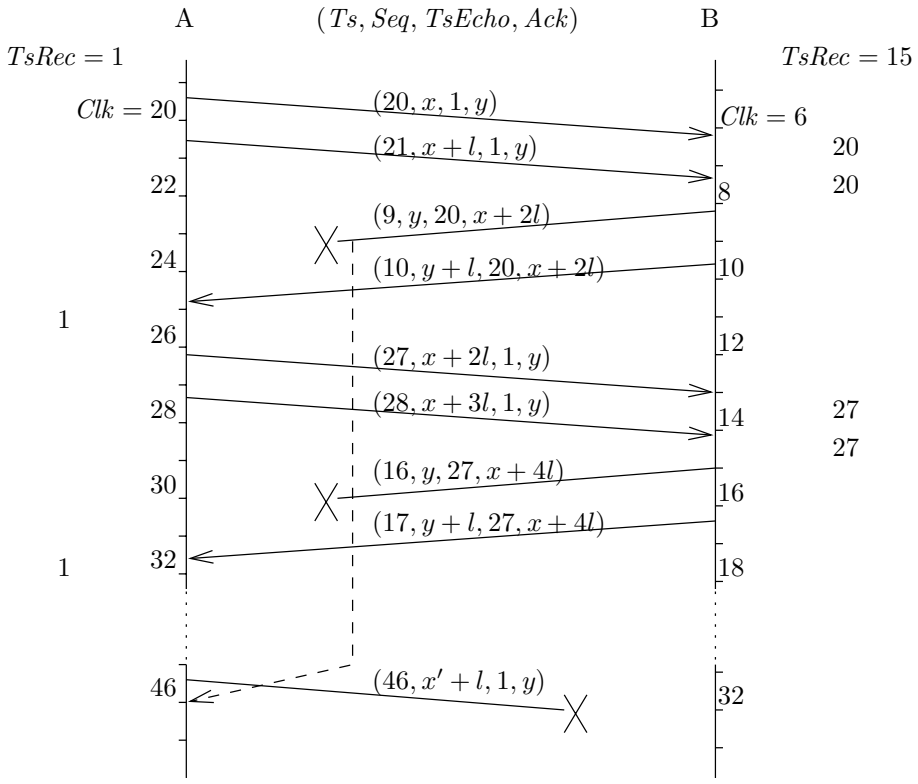


Figure 3.6: *Possible problem with the PAWS mechanism: incorrect validation of acknowledgments.*

situation, A will not be able to update its most recent timestamp $TsRec$, thus a duplicate ack appearing in the wrong moment can be accepted by A.

It is apparent that such a situation is not very likely to occur in practice. A practical retransmission algorithm will not retransmit multiple packets in face of persistent losses. Also, the chance of such regular packet loss pattern should be very low, although it may happen in case of some network malfunction. Still, it would not be wise to include this sort of assumptions into the formal model. If we assume that the transport entities may implement any retransmission strategy and network losses can occur in an arbitrary pattern, then the above described problem indeed exists.

The fundamental problem with PAWS as specified in [JBB92] is that plain TCP is a symmetric protocol, but PAWS is not. PAWS is asymmetric because $TsRec$, the protocol variable which is used to protect against wrapped sequence numbers (both data and acks), is updated only on the advance of data sequence numbers. If we use $TsRec$ only, we have to bind its updating algorithm either to the advance of data sequence numbers (forward direction) or to the advance of acknowledgments (reverse direction). The example in Figure 3.6 above demonstrates that the advance of the sequence numbers in the two directions is largely independent, therefore we cannot have a single variable to protect against the reuse of sequence numbers in both directions.

program *TS*;
Packet type definitions in Figure 3.8;
Variable definitions and initial conditions in Figure 3.9;
Sender process events in Figure 3.10 and 3.11;
Receiver process events in Figure 3.12.

Figure 3.7: *PAWS: program skeleton*

This observation leads to our simple solution. We make PAWS symmetric by introducing another protocol variable, the most recent echoed timestamp *TsEchoRec*. This variable has the same role for the validation of acks as $R.TsRec$ is for the validation of data.

Let us notice that with our modification the validation of the data and acknowledgment information become fully independent of each other. Because of this, the case of duplex data transfer can be simply considered as the combination of two simplex data transfer protocols. Therefore, just as in the case of the plain sliding-window protocol, it is sufficient to verify the simplex version of the protocol.

3.3.2 Protocol specification

The formal specification of PAWS is based on the sliding-window protocol specification in Section 3.2.1. Because the modifications are distributed over the whole specification, we decided to list the full specification again in the figures, but only the additional mechanisms of PAWS will be explained.

Figure 3.7 gives the outline of the specification. The details are shown in several listings. The protocol uses three different packet types, *DATA*, *ACK*, and *SACK* as it is shown in Figure 3.8. These types are the same used by the sliding-window protocol, but each packet type has new header fields. Some of these header fields are auxiliary variables, i.e., they are needed only for the verification.

Packet types

Data packets carry a timestamp *Ts* and an auxiliary field *SnMax*, which gives the maximum sequence number ($Nxt - 1$) that could have been sent until the time of sending this packet. Cumulative and selective acks get identical new fields. *TsEcho* is the echoed timestamp. This is a non-auxiliary field. *EchoExp* is the time of expiry of the *TsRec* echoed in the packet. *SnMax* echoes the header field of the same name from the data packet from which the echoed timestamp is taken. The meaning of these auxiliary fields (*EchoExp* and *SnMax*) can be better understood after reading the specification of the sender and receiver processes.

```
type dataunit = "the type of data units";
     packtype = (DATA, ACK, SACK);
     packet = record of
                case Type: packtype of
                DATA:
                  Seq: int;
                  Len: [0 ... RW];
                  Ts: int;
                  SnMax: int;
                  Data: array [0 ... RW] of dataunit;
                ACK:
                  Ack: int;
                  Wnd: [0 ... RW];
                  TsEcho, EchoExp: int;
                  SnMax: int;
                SACK:
                  Ack: int;
                  Len: [0 ... RW];
                  TsEcho, EchoExp: int;
                  SnMax: int;
```

Figure 3.8: *PAWS: packet type definitions*

Variable definitions

The variable definitions are shown in Figure 3.9. The sender process has two new auxiliary history variables, *TsMin* and *TsMax*. $TsMin[i]$ gives a lower bound of the timestamp that has been used in any packet carrying data units with sequence numbers not more than i. In other words, $P.Ts \geq TsMin[i]$ holds if $P.SnMax = i$. Similarly, $TsMax[i]$ gives the upper bound of these timestamps.

The arrays of epoch variables t_S and t_C record the time when a particular data unit was produced, and when the value of the clock *Clk* was incremented, respectively. The variable *Clk* holds the local notion of time. *TsEchoRec* is the most recent echoed timestamp received from ack packets. The role of the boolean variable *EchoRecOld* is to indicate when *TsEchoRec* has not been updated for a long time. This is needed to avoid the misinterpretation of *TsEchoRec* in modulo-N implementations when *TsEchoRec* becomes too old with respect to *Clk*. *Clk*, *TsEchoRec*, and *EchoRecOld* are all non-auxiliary variables.

The receiver has a new array of epoch variables t_C which records the times when the local clock *Clk* is incremented. The role of the *LAck* and *TsRec* variables was explained earlier during the informal description of the PAWS algorithm. *TsRec*, similarly to *TsEchoRec*

channel $C_{S,R}$, $C_{R,S}$;

process S;
  var *Source*: **array** $[0 \ldots \infty]$ **of** *dataunit* $\cup$ `empty`;
    Acked: **array** $[0 \ldots \infty]$ **of bool**;
    $TsMin, TsMax$: **array** $[0 \ldots \infty]$ **of int** $\cup$ `empty`;
    t_S, t_C: **array** $[0 \ldots \infty]$ **of epoch**;
    Una, *Nxt*: **int**;
    Wnd: $[0 \ldots RW]$;
    Clk, *TsEchoRec*: **int**;
    EchoRecOld: **bool**;
    SnMax: **int**;
  See Figure 3.10 and 3.11 for the events of process S.

process R;
  var *Sink*: **array** $[0 \ldots \infty]$ **of** *dataunit* $\cup$ `empty`;
    t_C: **array** $[0 \ldots \infty]$ **of epoch**;
    Nxt: **int**;
    Wnd: $[0 \ldots RW]$;
    LAck, *SnMax*: **int**;
    Clk, *TsRec*, *TsExp*: **int**;
    RecOld: **bool**;
  See Figure 3.12 for the events of process R.

init $S.Source[0 \ldots \infty] = R.Sink[0 \ldots \infty] = (\texttt{empty}, \texttt{empty}, \ldots)$;
  $S.Acked[0 \ldots \infty] = (\texttt{false}, \texttt{false}, \ldots)$;
  $S.TsMin[0 \ldots \infty] = S.TsMin[0 \ldots \infty] = (\texttt{empty}, \texttt{empty}, \ldots)$;
  $S.t_S[0 \ldots \infty] = S.t_C[0 \ldots \infty] = R.t_C[0 \ldots \infty] = (\lambda, \lambda, \ldots)$;
  $S.Una = S.Nxt = R.Nxt = S.Wnd = R.Wnd = 0$;
  $S.Clk = S.TsEchoRec = R.Clk = R.TsRec = R.TsExp = 0$;
  $S.EchoRecOld = R.RecOld = \texttt{false}$;
  $S.SnMax = R.SnMax = R.LAck = 0$;

Figure 3.9: *PAWS: variable definitions and initial conditions*

in the sender, must be aged to avoid the misinterpretation of timestamps when *TsRec* becomes old. *TsExp* is the value of the local, i.e., the receiver's clock at the time when *TsRec* was last updated. *RecOld* is boolean variable which is set to `true` when *TsRec* is aged. The auxiliary variable *SnMax* stores the header field of the same name from the last data packet which updated *TsRec*.

There are two new constants in the specification for the aging of timestamp-related variables. TW_S and TW_R are required for the aging of $S.TsEchoRec$ and $R.TsRec$, respectively. When the difference between $S.TsEchoRec$ and the current time $S.Clk$

event *ClockTick*;
  when `true` **do**
    $t_C[Clk] := \tau$; $Clk := Clk + 1$;
    if $\neg EchoRecOld \wedge Clk = TsEchoRec + TW_S$ **then**
      $EchoRecOld :=$ `true`;

event *Accept*(d);
  when $Nxt < Una + Wnd$ **do**
    $Source[Nxt] := d$; $t_S[Nxt] := \tau$;
    $TsMin[Nxt] := Clk$; $TsMax[Nxt] := Clk$;
    $Nxt := Nxt + 1$;

event *SendD*;
  var D: *packet*;
  when $Una < Nxt$ **do**
    Select i, l such that $Una \leq i < i + l \leq Nxt$;
    $D.Type := DATA$; $D.Seq := i$; $D.Len := l$;
    $D.Ts := Clk$; $D.SnMax := Nxt - 1$; $TsMax[Nxt - 1] := Clk$;
    $D.Data := Source[i \ldots i + l - 1]$;
    send($C_{S,R}, D$);

Figure 3.10: *PAWS: sender events, part 1*

becomes TW_S clock ticks, *S.EchoRecOld* is set to `true`. Similarly, if the receiver cannot update *TsRec* for more than TW_R clock ticks as measured on its local clock *R.Clk*, then *R.RecOld* is set to `true`.

The initial state is again self-explanatory. The state variables are equal to 0 and the history variables are empty.

Sender events

The specification of the sender events can be found in Figures 3.10 and 3.11. The *ClockTick* event records in the epoch variable t_C the end of the current clock tick and then increments the clock. The recent echoed timestamp *TsEchoRec* is aged when it gets older that $Clk - TW_S$ by setting *EchoRecOld* to `true`. The guard of the clock event is always true but when proving the CI conditions during the verification, we will add a real-time assumption which specifies the clock-rate.

The only modification in the *Accept* event is the update of the *TsMin* and *TsMax* history variables. $TsMin[n]$ and $TsMax[n]$ record the lower and upper limit of timestamps assigned to data packets when the right edge of the send window ($Nxt - 1$) was n.

```
event SendP;
    var D: packet;
    when Wnd = 0 do
        D.Type := DATA; D.Seq := Nxt; D.Len := 0;
        D.Ts := Clk; D.SnMax := Nxt; TsMax[Nxt − 1] := Clk;
        send(C_{S,R}, D);

event RecACK;
    var A: packet;
    when head(C_{R,S}) = A ∧ A.Type = ACK do
        if (EchoRecOld ∨ A.TsEcho ≥ TsEchoRec) ∧ (Una < A.Ack ≤ Nxt) then
            Acked[Una ... A.Ack − 1] := true;
            Una := A.Ack; Wnd := A.Wnd;
            TsEchoRec := A.TsEcho; EchoRecOld := false; SnMax := A.SnMax;
        else if (EchoRecOld ∨ A.TsEcho ≥ TsEchoRec) ∧ Una = A.Ack then
            Wnd := max(Wnd, A.Wnd); TsEchoRec := A.TsEcho; EchoRecOld := false;

event RecSACK;
    var A: packet;
    when head(C_{R,S}) = A ∧ A.Type = SACK do
        if (EchoRecOld ∨ A.TsEcho ≥ TsEchoRec)∧
                (Una < A.Ack < A.Ack + A.Len ≤ Nxt) then
            Acked[A.Ack ... A.Ack + A.Len − 1] := true;
```

Figure 3.11: *PAWS: sender events, part 2*

The logic of the *SendD* event does not change, but it must now fill in the new fields in the packet header. The auxiliary variable $TsMax[Nxt-1]$ is also updated from the local clock. The *SendP* event is not completely new in this specification although it did not appear explicitly in the sliding-window protocol. Note however, that this event was part of one of the three possible fairness assumptions. Because, as we will see later, in case of PAWS only one fairness assumption is feasible which assumes this event, it is included now in the formal specification. The event is enabled when the send window is empty and it generates zero-length data packets to probe the receiver until an acknowledgment with a non-zero window is received.

In the *RecACK* and *RecSACK* events the validation procedure of the sliding-window protocol is augmented with a check of the echoed timestamp in the packet against *TsEchoRec* which is the most recent echo from earlier acks.

event *ClockTick*;
  when `true` **do**
    $t_C[Clk] := \tau$;
    $Clk := Clk + 1$;
    if $\neg RecOld \wedge Clk = TsExp + TW_R$ **then**
      $RecOld :=$ `true`;

event *ExpandWindow*;
  when $Wnd < RW$ **do** $Wnd := Wnd + 1$;

event *RecD*;
  var D: *packet*;
  when head$(C_{S,R}) = D \wedge D.Type = DATA$ **do**
    if $(RecOld \vee D.Ts > TsRec) \wedge D.Seq < LAck$ **then**
      $TsRec := D.Ts$; $RecOld :=$ `false`; $TsExp := Clk$; $SnMax := D.SnMax$;
    if $(RecOld \vee D.Ts \geq TsRec) \wedge (Nxt < D.Seq + D.Len \leq Nxt + Wnd)$ **then**
      $j := \max(Nxt, D.Seq)$;
      $Sink[j \ldots D.Seq + D.Len - 1] := D.Data$;
      while $R.Wnd > 0 \wedge Sink[Nxt] \neq$ `empty` **do**
        $t_R[Nxt] := Nxt + 1$; $Wnd := Wnd - 1$;

event *SendACK*;
  var A: *packet*;
  when $\neg RecOld$ **do**
    $A.Type := ACK$; $A.Ack := Nxt$; $A.Wnd := Wnd$;
    $A.TsEcho := TsRec$; $A.SnMax := SnMax$; $A.EchoExp := TsExp$;
    send$(C_{R,S}, A)$;

event *SendSACK*;
  var A: *packet*;
  when $\neg RecOld \wedge$ 'there exists i, l such that $Nxt < i < i + l \leq Nxt + Wnd$,
      $Sink[i-1] =$ `empty` and $Sink[i \ldots i + l - 1] \neq$ `empty`' **do**
    $A.Type := SACK$; $A.Ack := i$; $A.Len := l$;
    $A.TsEcho := TsRec$; $A.SnMax := SnMax$; $A.EchoExp := TsExp$;
    send$(C_{R,S}, A)$;

Figure 3.12: *PAWS: receiver events*

Receiver events

The receiver events are listed in Figure 3.12. The *ClockTick* event uses the same logic as the *ClockTick* event in the sender. The *ExpandWindow* event is identical to the one in the sliding-window protocol.

The first 'if' statement of the *RecD* event handles the update of *TsRec*. *TsExp* records the value of the receiver's clock when *TsRec* was last updated. The auxiliary variable *SnMax* contains the right edge of the send window at the moment when the data packet used to update *TsRec* was sent. The second 'if' statement checks whether the received packet contains acceptable data. Data inside the receive window is acceptable if the timestamp of the packet is not older than *TsRec*. The processing of data in the second 'if' construct is identical to the body of the *RecD* event in the sliding-window protocol.

The *SendACK* and *SendSACK* events generate an ack when they are triggered. The sending of acks is disabled when the recent timestamp becomes too old to prevent the ambiguity caused by wrapped timestamps.

3.3.3 Safety and progress of the unbounded protocol

The strategy for verifying PAWS is similar to the strategy used for the verification of the plain sliding-window (SW) protocol. First we prove that the protocol satisfies the desired properties assuming unbounded sequence numbers and timestamps. Then the CI conditions are formulated and their invariance is proven which indicates that the protocol can be implemented using modulo-N representation for the state variables.

Safety

The proof that PAWS satisfies the safety properties (3.1), (3.2) is based on the safety verification of the SW protocol. The safety of PAWS follows directly from the safety of the SW protocol because PAWS can be considered as a specific version of the SW protocol. In other words, any packet that is accepted by PAWS would also be accepted by the SW protocol. Since the safety properties mean that "nothing bad can happen," the safety of the SW protocol implies the safety of PAWS.

To prove that this relation indeed exists between PAWS and the SW protocol, we show that the protocols satisfy three conditions:

1. The state variable set of PAWS is a superset of the of the state variable set of SW.
2. The initial condition of PAWS implies the initial condition of SW.
3. Each computation of PAWS can be mapped to a computation of the SW protocol.

The first condition holds if we do not count auxiliary variables, e.g., $S.t_A$ is a variable in SW but not in PAWS. Note that we can safely omit the auxiliary variables because by their definition they cannot influence the behavior of the protocol. Each state s of PAWS can be mapped to the state $\tilde{s}$ of SW obtained from s by ignoring the values of state variables not in SW.

The second condition also holds which is easy to check.

To check that the third condition is satisfied, consider the following. The new events in PAWS, *S.ClockTick* and *R.ClockTick* do not alter any of the state variables present in SW. Any occurrence of such events in a computation of PAWS can be mapped to an imaginary idle event. The events *S.Accept*, *S.SendD*, *S.SendP*, *R.ExpandWindow*, *R.SendACK*, and *R.SendSACK* of PAWS are identical to those in SW if we omit the new variables. That is, any occurrence of these events in a computation of PAWS maps to the same event of SW.

Finally, the events *S.RecACK*, *S.RecSACK*, and *R.RecD* which deal with the validation of received packets have to be considered. Each of these receive events are enabled when a packet of the appropriate type is at the head of the channel. The body of these events contains an 'if' construct in order to validate the received packet. If the packet is acceptable, then it is processed otherwise there is no change in the state of the system except the removal of the packet from the channel. The body of the 'if' constructs is identical in the two protocols considering only the variables of SW. The difference is that there is an extra condition in the PAWS events for the validation of the packet using the new state variables. Therefore, the occurrence of the receive events in a computation of PAWS can be mapped to the same receive event of SW if the extra condition evaluates to true. Otherwise, the occurrence of the PAWS receive event maps to a packet drop of the appropriate channel in SW. Note that this is a valid state transition in the SW protocol because channel events are enabled when there are packets in transit.

Thus we produced a mapping which assigns a valid computation of the SW protocol to every computation of PAWS. Because the SW protocol satisfies the desired safety properties, this mapping implies that PAWS satisfies these safety properties as well.

Progress

The desired progress properties of PAWS are defined by assertions (3.13–3.15). The same desired properties were assumed for the SW protocol. Similarly to the SW protocol, we have to make some liveness assumptions about the entities as well. The basic difference is that in case of PAWS there is only one set of liveness assumptions instead of three.

Let us notice that the *R.SendACK* event is not always enabled in PAWS. This is necessary to avoid confusion from wrapped timestamps when *R.TsRec* has not been updated for a certain interval (denoted by TW_R is the specs). If there is a temporary channel failure longer than this timeout period, then the system enters a state in which only the sender is allowed to generate packets. Thus, in order to assure progress, the sender has to actively probe the receiver when it expects more information from the receiver. This requirement can be formulated in the following set of liveness assumptions:

$$\begin{aligned} &S.Nxt > S.Una = n \wedge (\#D : n \in [D.Seq \ldots D.Seq + D.Len - 1]) = k \rightsquigarrow \\ &\quad \rightsquigarrow \quad S.Una > n \vee (\#D : n \in [D.Seq \ldots D.Seq + D.Len - 1]) > k \end{aligned} \tag{3.27}$$

$$S.Wnd = 0 \wedge (\#D : D.Len = 0) = k \leadsto$$
$$\leadsto \quad S.Wnd > 0 \vee (\#D : D.Len = 0) > k \tag{3.28}$$
$$(\neg enabled(R.SendACK) \leadsto enabled(R.SendACK)) \Rightarrow$$
$$\Rightarrow \quad ((\#CA) = k \leadsto (\#CA) = k + 1) \tag{3.29}$$

These liveness assumptions correspond to the second set of liveness assumptions for SW defined by (3.16), (3.18), and (3.19). The major difference is in the assertion (3.29). This assertion requires that if the *R.SendSACK* event is enabled infinitely often, then cumulative acks are sent infinitely often. In other words, this is a strong fairness requirement for the event *R.SendACK*. In case of the SW protocol, the corresponding requirement (assertion (3.19)) is somewhat weaker. For PAWS, we have to consider that the *R.SendSACK* event is not continuously enabled, which necessitates the stronger fairness assumption. In practice, however, it is still not a very strict requirement because, as we will see later, the time period *R.SendSACK* remains enabled after the reception of a valid data packet is usually in the order of days. Therefore the requirement that a receiver must send at least one ack in every such period does not constrain implementations too much.

Assertion (3.28) is equivalent to weak-fairness for the *S.SendP* event, while assertion (3.27) is weak-fairness for the *S.SendD* event with the restriction that the lowest unacknowledged data word has to be eventually retransmitted. Apart from these fairness assumptions, we also assume that both channels are fair (see Section 2.3.3).

Similarly to the proof of the invariants, the progress properties are proven by proving a series of progress properties which finally lead to the desired properties. Only an outline of the proof is given here in the thesis, further details can be found in Appendix A.2 and [Olá95]. The important properties derived during the proof are given below with some indication of their meaning and the proof rules which can be used to prove these assertions.

$$S.Nxt > R.Nxt = S.Una = n \leadsto R.Nxt > n \tag{3.30}$$

$$S.Una + S.Wnd = S.Una = n \wedge R.Wnd > 0 \leadsto S.Una + S.Wnd > n \tag{3.31}$$

Assertion (3.30) and (3.31) are two restricted versions of (3.13) and (3.15), respectively. (3.30) says that if every data word that is received is also acknowledged and the sender has some outstanding data, then the receiver eventually receives more data. (3.30) is expected to hold because of the liveness assumption (3.27) and the channel fairness assumption. (3.31) says that if the sender has a zero window and the receiver has space for more data then eventually the sender window will open up. (3.31) will be used to prove (3.15).

$$S.Una = n \wedge S.Clk = m \rightarrow (D \in C_{S,R} \wedge D.Ts > m \Rightarrow D.Seq \geq n) \tag{3.32}$$

$$S.Nxt > S.Una = n \wedge S.Clk = m \rightarrow$$
$$\rightarrow \quad (D \in C_{S,R} \wedge D.Seq = n \wedge D.Ts > m \Rightarrow D.Len > 0) \tag{3.33}$$

$$S.Nxt > R.Nxt = S.Una = n \wedge S.Clk = m \rightarrow$$
$$\rightarrow \quad (R.Nxt = n \Rightarrow R.TsRec \leq m) \tag{3.34}$$

(3.32)–(3.34) are used in the proof of (3.30). (3.32) states that once $S.Una = n$ and $S.Clk = m$ then any packet sent in a later state has a sequence number that is at least n. We expect this to hold because of the definition of the $S.SendD$ and $S.SendP$ events and the monotonicity of $S.Una$ and $S.Clk$. (3.33) says that once $S.Nxt > S.Una = n$ holds, any data packet sent from that time on with a sequence number equal to n must have non-zero length. (3.33) holds because zero-length packets can only be sent when $S.Wnd = 0$ which implies $S.Nxt = S.Una$ and because of the monotonicity of $S.Nxt$. (3.34) asserts that if (i) the sender has data to send, and (ii) all the in-sequence data received by the receiver has also been acknowledged, then $R.TsRec$ cannot grow unless new data is accepted by the receiver. Assertion (3.34) is a precondition of (3.30) because it assures that data packets with new data will indeed be accepted by the $R.RecD$ event.

$$R.Nxt > S.Una = n \wedge S.Clk \geq m \rightsquigarrow$$
$$\rightsquigarrow \quad S.Una > n \vee (\neg R.RecOld \wedge R.TsRec \geq m) \tag{3.35}$$

$$R.Nxt = n \wedge R.TsRec = m \rightarrow (A \in C_{R,S} \wedge A.TsEcho > m \Rightarrow A.Ack \geq n) \tag{3.36}$$

$$R.Nxt > S.Una = n \wedge S.Clk = m \rightarrow (S.Una = n \Rightarrow S.TsEchoRec \leq m) \tag{3.37}$$

With the help of (3.35–3.37), the desired property (3.14) can be proven. (3.35) states that if some data is not yet acknowledged then either the data will be acked or the receiver's recent timestamp becomes "new." New means here that $\neg R.RecOld$ holds and thus $R.SendACK$ is enabled. We expect (3.35) to hold because the sender retransmits unacknowledged data as formulated in the liveness assumption (3.27). (3.36) is very similar to (3.32) but it is for the reverse channel. It holds because of the monotonicity of the variables involved. (3.37) is the equivalent of (3.34) for the reverse channel. (3.37) is needed to prove that if the sender gets an acknowledgment with $A.Ack > n$ and $A.TsEcho \geq m$, then its reception will establish $S.Una > n$.

$$S.Wnd = 0 \wedge S.Clk \geq m \rightsquigarrow S.Wnd > 0 \vee (\neg R.RecOld \wedge R.TsRec \geq m) \tag{3.38}$$

$$R.Nxt + R.Wnd > n \wedge R.TsRec = m \rightarrow$$
$$\rightarrow \quad (A \in C_{R,S} \wedge A.TsEcho > m \Rightarrow A.Ack + A.Wnd > n) \tag{3.39}$$

$$S.Una + S.Wnd = S.Una = n \wedge R.Wnd > 0 \wedge S.Clk = m \rightarrow$$
$$\rightarrow \quad (S.Una + S.Wnd = n \Rightarrow S.TsEchoRec \leq m) \tag{3.40}$$

The assertions (3.38)–(3.40) are needed for proving (3.31). (3.38) asserts that if the sender window is closed then either the window will open up or the receiver's recent timestamp becomes new. The assertion is analogous to (3.35). The assertion is expected to hold because of the liveness assumption (3.28). Assertion (3.39) is the analogy of (3.33) and (3.36) for the right edge of the receive window. It asserts that if the right edge of the window is greater than n, then the right edge of the window in any subsequent ack packet is at least as high. (3.39) holds because of the monotonicity of the variables involved. (3.40) is analogous to the assertions (3.34) and (3.37). It asserts that if the sender has no outstanding data and a zero send window, then $S.TsEchoRec$ cannot grow unless the right edge of the send window is advanced. This property is needed to prove that a window update is eventually accepted by the $S.RecACK$ event.

3.3.4 Correct interpretation conditions

Formulating the CI conditions

Similarly to the SW protocol, we have to formulate the CI conditions in order to prove that the specification can be implemented using finite representation for the state variables. In PAWS, there are two different sorts of state variables: one sort to represent sequencing variables and another to represent timestamps and clocks. The constants N_S and N_C denote the size of the sequence space, and the size of the timestamp space, respectively. That is, $S.Una$, $R.Nxt$, $D.Seq$,... are implemented as modulo-N_S variables, $S.Clk$, $R.TsRec$, $A.TsEcho$,... are implemented as modulo-N_C variables.

Two sorts of CI conditions will be formulated for PAWS:

- CI conditions to assure the correct interpretation of sequence numbers assuming the correct interpretation of timestamps;
- CI conditions to assure the correct interpretation of timestamps.

$$\Box(D \in C_{S,R} \wedge (R.RecOld \vee D.Ts \geq R.TsRec) \Rightarrow$$
$$\Rightarrow \quad D.Seq \geq R.Nxt + RW - N_S + 1) \tag{3.41}$$

$$\Box(A \in C_{R,S} \wedge (S.EchoRecOld \vee A.TsEcho \geq S.TsEchoRec) \Rightarrow \\ \Rightarrow \quad A.Ack \geq S.Nxt - N_S + 1) \tag{3.42}$$

Assertions (3.41) and (3.42) are the conditions for the correct interpretation of sequence numbers in the $R.RecD$ and $S.RecACK$, $S.RecSACK$ events respectively. These CI conditions are almost equivalent to the CI conditions (3.21) and (3.22) of the SW protocol. The only difference is that the correct interpretation of sequence numbers has to be assured only in the case when the packet has passed the timestamp validation. This is reflected by the appearance of the extra condition in the antecedent of the implications. In other words, we could weaken the CI conditions for sequence numbers because of the extra protection provided by the timestamps.

$$(\exists K_R : \Box(D \in C_{S,R} \wedge \neg R.RecOld \Rightarrow \\ \Rightarrow \quad R.TsRec + K_R \geq D.Ts \geq R.TsRec + K_R - N_C + 1)) \tag{3.43}$$

$$\Box(A \in C_{R,S} \wedge \neg S.EchoRecOld \Rightarrow A.TsEcho \geq S.Clk - N_C + 1) \tag{3.44}$$

Assertions (3.43) and (3.44) are the CI conditions corresponding to the $R.RecD$ and $S.RecACK$, $R.RecSACK$ events, respectively. These conditions are necessary to assure the correct interpretation of the timestamps in the packets.

Assertion (3.43) is the CI condition for the comparisons '$D.Ts \geq R.TsRec$' and '$D.Ts > R.TsRec$' in the $R.RecD$ event. It was obtained by substituting the corresponding variables directly into the definition of CI conditions (3.5). The value of constant K_R will be determined during the verification.

$$\Box(\neg R.RecOld \Rightarrow R.TsExp \geq R.Clk - TW_R + 1) \tag{3.45}$$

$$\Box(\neg S.EchoRecOld \Rightarrow S.TsEchoRec \geq S.Clk - TW_S + 1) \tag{3.46}$$

Assertions (3.45) and (3.46) assure the correct interpretation of the variables $R.TsExp$ and $S.TsEchoRec$ in the $R.ClockTick$ and $S.ClockTick$ events, respectively. It is easy to show that the above assertions hold. Furthermore, they imply the CI conditions that can be obtained from the definition (3.5) if we assume that the inequalities $TW_R < N_C$ and $TW_S < N_C$ are satisfied by the constants.

Real-time assumptions

We have to make assumptions about the real-time properties of the clocks in the entities for the verification of the CI properties. These assumptions extend the two real-time

assumptions about the maximum transmission rate B and the maximum packet lifetime L made during the verification of the SW protocol.

$$\Box(S.t_C[n-1] \neq \lambda \wedge S.t_C[n] \neq \lambda \Rightarrow S.t_C[n] - S.t_C[n-1] > \gamma_S) \tag{3.47}$$

$$\Box(S.t_C[n-1] \neq \lambda \wedge S.t_C[n] = \lambda \Rightarrow \tau - S.t_C[n-1] < \Gamma_S) \tag{3.48}$$

$$\Box(R.t_C[n-1] \neq \lambda \wedge R.t_C[n] \neq \lambda \Rightarrow R.t_C[n] - R.t_C[n-1] > \gamma_R) \tag{3.49}$$

$$\Box(R.t_C[n-1] \neq \lambda \wedge R.t_C[n] = \lambda \Rightarrow \tau - R.t_C[n-1] < \Gamma_R) \tag{3.50}$$

In these assertions, γ_S and Γ_S denote the minimum and the maximum time between two *S.ClockTick* events. The clock rate r thus satisfies $1/\Gamma_S \leq r \leq 1/\gamma_S$. The real-time properties of the other clock are defined in the same manner.

Assertions from the verification

Some of the key assertions from the verification of the CI conditions are explained below.

$$\Box(0 \leq n < S.Nxt \Rightarrow S.TsMin[n] \leq S.TsMax[n] \leq S.Clk) \tag{3.51}$$

$$\Box(0 < n < S.Nxt \Rightarrow S.TsMax[n-1] \leq S.TsMin[n]) \tag{3.52}$$

$$\Box(\lceil \Gamma_S \cdot B \rceil \leq n < S.Nxt \Rightarrow S.TsMin[n - \lceil \Gamma_S \cdot B \rceil] < S.TsMin[n]) \tag{3.53}$$

Assertions (3.51)–(3.53) list properties of the history variables $S.TsMin$ and $S.TsMax$. These properties are consequences of the monotonicity of $S.Clk$ and $S.Nxt$. (3.53) creates the connection between timestamps and sequence numbers by stating that no more than $\lceil \Gamma_S \cdot B \rceil$ sequence numbers can be consumed in one clock tick.

$$\Box(D \in C_{S,R} \Rightarrow S.TsMin[D.SnMax] \leq D.Ts \leq S.TsMax[D.SnMax]) \tag{3.54}$$

$$\Box(S.TsMin[R.SnMax] \leq R.TsRec \leq S.TsMax[R.SnMax]) \tag{3.55}$$

$$\Box(A \in C_{R,S} \Rightarrow S.TsMin[A.SnMax] \leq A.TsEcho \leq S.TsMax[A.SnMax]) \tag{3.56}$$

$$\Box S.TsMin[S.SnMax] \leq S.TsEchoRec \leq S.TsMax[S.SnMax]) \tag{3.57}$$

Assertions (3.54)–(3.57) define the meaning of $S.TsMin$ and $S.TsMax$ with respect to the various timestamps carried in the packets and stored in the protocol entities. (3.54), for example, provides bounds for the timestamp in data packets.

$$\Box(R.Nxt - RW \leq R.SnMax \leq S.Nxt - 1) \tag{3.58}$$

$$\Box(S.Una - RW \leq S.SnMax \leq S.Nxt - 1) \tag{3.59}$$

Assertions (3.58) and (3.59) play a key role in proving the invariance of the CI conditions (3.41) and (3.42). These assertions provide the relation between the auxiliary variable $SnMax$ and the send/receive window, respectively.

Using the real-time assumptions, the assertions above and some more listed in Appendix A.1, one can prove that the CI conditions of PAWS are invariant provided that the protocol parameters satisfy the following requirements:

$$N_S \geq 3RW + \lceil \Gamma_S \cdot B \rceil \tag{3.60}$$

$$N_C \geq \left\lceil \frac{L + TW_R \cdot \Gamma_R}{\gamma_S} \right\rceil + \left\lceil \frac{L}{\gamma_S} \right\rceil + 3 \tag{3.61}$$

$$TW_R \geq \left\lceil \frac{L}{\gamma_R} \right\rceil + 1 \tag{3.62}$$

$$TW_S \geq \left\lceil \frac{2L + TW_R \cdot \Gamma_R}{\gamma_S} \right\rceil + 1 \tag{3.63}$$

(3.60) is needed for the correct interpretation of sequence numbers because it is the precondition of the invariance of CI conditions (3.41) and (3.42). The other three inequalities are needed for the correct interpretation of timestamps. The above conditions are believed to be tight, i.e., the violation of any of them could cause the protocol to malfunction. An informal argument about the tightness of these conditions can be found [Olá95].

3.4 SNR: a periodic state-exchange protocol

SNR is a light-weight data transfer protocol which was designed for high-speed networks [NRS90]. The protocol is based on the idea of periodic state exchange. This is achieved by having the receiver send its full state to the sender at regular intervals. In contrast, other

protocols send state information only in the case of some state changes or occurrence of other events.

Sending *full* state information facilitates a selective retransmission strategy. Because the sender has exact information about the arrival and loss of data units at the receiver, it knows exactly which units must be retransmitted. Sending the state information *regularly* allows the sender to use a simple retransmission scheme: the loss of state messages is corrected automatically at the arrival of the next state message and the arrival of state messages can also be used to count retransmission timeouts.

Due to the periodic state exchange, the protocol generates network load even if there is no user traffic. This can be negligible in a high speed network, but could turn out to be prohibitive in a slow network where efficient usage of the available bandwidth is crucial. Therefore the protocol is better suited for high speed environments. More details about the idea of periodic state exchange and its performance characteristics can be found in [NRS90] and [DJNS93], respectively.

The formal specification and verification of a data transfer protocol based on the idea of periodic state exchange was presented in [GNS95]. The authors prove that the protocol behaves correctly if some constraints are respected involving the rate of sending state messages, the size of buffers and the range of sequence numbers. The protocol is included in the thesis for the following reasons:

- Some of its mechanisms are apparently different from the mechanisms used in the sliding-window protocols discussed so far in Section 3.2 and 3.3. Therefore the inclusion of SNR widens the range of data transfer protocols that are covered by the thesis and it allows for the comparison with the other data transfer protocols.
- Some improvements to SNR are also presented.

The specification in [GNS95] uses unbounded sequence numbers in the state messages. Although it is often a useful abstraction, such sequence numbers cannot be used in implementations. The authors suggest to use a large number space for these sequence numbers and to reset the protocol every time when they wrap around. A protocol reset consists of an idle period for the maximum packet lifetime to allow all packets to disappear from the network. A modification is proposed in Section 3.4.3 which eliminates the need for these costly protocol resets.

The constraints derived in [GNS95] may also limit the practical value of the protocol in some environments. In Section 3.4.4 we show that the lower bound on the retransmission timeout enforced by the specification in [GNS95] becomes too high in a network with no tight bound on the maximum packet lifetime. As an alternative solution, the idea of periodic state exchange will be incorporated into the sliding-window protocol specification discussed in Section 3.2. Some results of this Section have been published in [OHdG96a, OHdG97].

program *SNR*;
type *dataunit* = "the type of data units";
dt = **record of**
i: $0 \ldots n-1$;
data: *dataunit*;
st = **record of**
b: **array** $[0 \ldots n-1]$ **of bool**;
r, t: $0 \ldots n-1$;
k, r_u: **int**;

channel $C_{p,q}$, $C_{q,p}$; { no duplicates, only drop or reorder }

process P;
var *Sink*: **array** $[0 \ldots \infty]$ **of** *dataunit* $\cup$ `empty`;
rcvd: **array** $[0 \ldots n-1]$ **of bool**;
pr, pt: $0 \ldots n-1$;
pk, pr_u: **int**;
See Figure 3.14 for the events of process P.

process Q;
var *Source*: **array** $[0 \ldots \infty]$ **of** *dataunit* $\cup$ `empty`;
Acked: **array** $[0 \ldots \infty]$ **of bool**;
src: **array** $[0 \ldots n-1]$ **of** *dataunit*;
ackd: **array** $[0 \ldots n-1]$ **of bool**;
count: **array** $[0 \ldots n-1]$ **of** $0 \ldots m-1$;
qr, qs, qt: $0 \ldots n-1$;
qk, qr_u: **int**;
See Figure 3.15 for the events of process Q.

init $(\forall i : Q.Source[i] = P.Sink[i] = \texttt{empty} \wedge \neg Q.Acked[i]) \wedge$
$P.pr = Q.qr = Q.qs = 0 \wedge P.pt = Q.qt = 1 \wedge P.pk = Q.qk = 0 \wedge$
$P.pr\_u = Q.qr\_u = 0 \wedge$
$(\forall i \in [0 \ldots n-1] : \neg P.rcvd[i])$;

Figure 3.13: *SNR protocol: main part*

3.4.1 Specification

To explain our modifications, we start with the description of the original specification in [GNS95]. The specification given here is functionally equivalent with the original, but it is written in our notation in order to be consistent with the other specifications in the thesis. The specification can be found in Figure 3.13, 3.14, and 3.15.

The basic idea of the protocol is simple. The receiver has a circular buffer where received data units are stored until they are removed by the host of the receiver. In the state messages, the receiver sends the current state of the buffer to the sender. From these messages the sender can deduce which data units must be resent because they were lost and they also inform the sender about the amount of new data that can be sent.

State variables

The size of the receive buffer is n. The receiver P maintains a boolean map of the receive buffer *rcvd*, and two modulo-n pointers pr and pt. These three variables represent the state of the circular buffer. The two pointers divide the buffer into two non-empty regions. The locations in the range $[pt \ldots pr -_n 1]$ are occupied by previously received data units that have not been consumed by the host yet, therefore no new data can be accepted in this region. All entries of *rcvd* in this range are set to `false`. The region $[pr \ldots pt -_n 1]$ is a "can-receive" region. The sender is allowed to send new data units in this region. An entry of *rcvd* in this region is `true` if the corresponding data unit has already been received.

The corresponding state variables in the sender's state are *ackd*, qr, and qt. These variables store the most recent values of the receiver's state variables *rcvd*, pr, and pt taken from the state messages sent by the receiver. There is one more non-auxiliary variable of the receiver pk which stores the sequence number to be sent on the state messages. This sequence number is unbounded so that the sender can always distinguish between old and new state messages. Similarly, the sender maintains the variable qk which stores the most recent sequence number from state messages. A newly received state message is accepted if and only if its sequence number is greater than qk.

The sender is allowed to send new data units in the range $[qr \ldots qt -_n 1]$. The variable qs is the sequence number of the next data unit to be sent. The sender uses the periodic state messages for triggering retransmissions. A data unit is retransmitted if m state messages have been received since its last transmission without acknowledging its receipt. This is implemented by keeping an array of counters $count[0 \ldots n-1]$, one for each entry in the circular buffer. When a state message is received, the counters corresponding to the range $[qr \ldots qs -_n 1]$ are incremented. If one of them reaches m, more precisely 0 because the counters are modulo-m numbers, then the data unit is retransmitted if the corresponding entry in *ackd* is `false`. The outstanding data units are stored in the buffer *src*.

Furthermore, both the sender and the receiver maintain some auxiliary variables. The history variables *Source*, *Sink*, and *Acked* are from the specification of the abstract data-transfer service (see Section 3.1 for details). There are two auxiliary variables, one at the receiver and another at the sender, which maintain the unbounded value of pr and qr, respectively. With the help of these variables the correspondence between the history variables and the non-auxiliary buffer variables can be expressed. The region $[qr \ldots qs -_n 1]$ of the sender buffers *src* and *ackd* are equal to the corresponding slices

starting at the index qr_u of the history variables *Source* and *Acked*, respectively. This can be expressed formally by the following two assertions:

$$\Box(\forall i \in [0 \ldots qs -_n qr -_n 1] : src[qr +_n i] = Source[qr\_u + i])$$

$$\Box(\forall i \in [0 \ldots qs -_n qr -_n 1] : ackd[qr +_n i] = Acked[qr\_u + i])$$

Similarly, for the receiver the following assertion holds:

$$\Box(\forall i \in [0 \ldots pt -_n pr -_n 1] : rcvd[pr +_n i] = (Sink[pr\_u + i] \neq \texttt{empty}))$$

Two different message formats are used by the protocol. Data messages which are sent from the sender process to the receiver have two fields: a modulo-n sequence number i and the data unit carried in the message *data*. Notice that in this protocol each data message carries only a single a data unit. The state messages which are generated by the receiver have five fields: b, r, t, k, and r_u. These fields carry the current value of the receiver state variables *rcvd*, pr, pt, pk, and pr_u, respectively.

The messages are sent over two channels: $C_{p,q}$ and $C_{q,p}$. These channels can drop, delay and reorder messages, but unlike the channels in the previous protocols they *do not create duplicates*. This is important because the presence of duplicates can lead to protocol hazards as it will be discussed below.

Receiver events

The receiver has four events which are shown in Figure 3.14. The number of the corresponding action in [GNS95] is show in comments at each event so that our specification can easily be compared to the original specification.

The *SendStatus* event assembles and sends a state message to the sender process. The enabling condition of this event is the abstract *timeout* predicate which indicates that the execution of the event is related to real-time. The exact meaning of this construct will be discussed later during the protocol verification. The event increments first the sequence number pk and then sends a state message S.

Data messages are received in the *RecvData* event. The corresponding bit in the receive buffer map is set to `true` and the the received data unit is stored in the history array *Sink*.

The event *AdvWin* increments pr provided that the lowermost dataunit in the 'can-receive' region of the circular buffer has already been received and there is space enough in the 'can-receive' region. Parallel to pr, this event also increments the auxiliary variable pr_u so that the assertion $\Box(pr = pr\_u \bmod n)$ remains valid.

event *SendStatus*; { *1* }
  var S: st;
  when *timeout* **do**
    $pk := pk + 1$;
    $S.b, S.r, S.t, S.k, S.r\_u := rcvd, pr, pt, pk, pr\_u$;
    send$(C_{p,q}, S)$;
event *RecvData*; { *2* }
  var D: dt;
  when **head**$(C_{q,p}) \neq$ `nil` **do**
    $D :=$ **recv**$(C_{q,p})$;
    $rcvd[D.i], Sink[pr\_u + (D.i -_n pr)] :=$ `true`, $D.data$;
event *AdvWin*; { *3* }
  when $rcvd[pr] \wedge pr +_n 1 \neq pt$ **do**
    $rcvd[pr] :=$ `false`;
    $pr, pr\_u := pr +_n 1, pr\_u + 1$;
event *DeliverData*; { *4* }
  when $pt +_n 1 \neq pr$ **do**
    $pt := pt +_n 1$;

Figure 3.14: *SNR protocol: receiver events*

The event *DeliverData* adds space to the top of the 'can-receive' region by incrementing pt provided that the 'cannot-receive' region does not become empty as the result of executing the event.

Sender events

The sender has only two events: *RecvStatus* and *SendData*. The first event contains the processing of incoming state messages. A state message S is accepted only if $S.k > qk$, otherwise the message is discarded. If the message is acceptable, the state variables of the sender are updated from the corresponding message fields. Then the history array *Acked* is updated with the new value of *ackd*. After that the counters belonging to the region $[qr \ldots qs -_n 1]$ are incremented. If any of them becomes 0 and the corresponding data unit is not yet acknowledged, then it is retransmitted.

New data can only be sent if qs is less than qt which is the upper edge of the 'can-send' region. In such a case, the *SendData* event records the new data unit in the send buffer *src* and in the history array *Source*. Then a data message is assembled and sent. Finally, the corresponding counter $count[qs]$ is reset and qs is incremented. This completes the description of the protocol events.

```
event RecvStatus;                                                    { *5* }
    var S: st;
        j : 0 ... n − 1;
    when head(C_{p,q}) ≠ nil do
        S := recv(C_{p,q});
        if S.k > qk then
            ackd, qr, qt, qk, qr_u := S.b, S.r, S.t, S.k, S.r_u;
            for j = 0 ... qs −_n qr −_n 1 do
                Acked[qr_u + j] := ackd[qr +_n j];
            for j = qr ... qs −_n 1 do
                count[j] := count[j] +_m 1;
                if count[j] = 0 ∧ ¬ackd[j] then
                    D.i, D.data := j, src[j];
                    send(C_{q,p}, D);
event SendData(d : dataunit);                                        { *6* }
    var D: dt;
    when qs +_n 1 ≠ qt do
        src[qs], Source[qr_u + (qs −_n qr)] := d, d;
        D.i, D.data := qs, src[qs];
        send(C_{q,p}, D);
        count[qs] := 0; qs := qs +_n 1;
```

Figure 3.15: *SNR protocol: sender events*

3.4.2 Verification and protocol properties

Both safety and progress properties of the protocol are verified in [GNS95]. The correctness of this protocol, just as the others discussed in the this thesis, depends on the real-time parameters of the system. The authors of [GNS95] used a different approach to deal with the real-time aspects during the verification.

Remember that the enabling condition of the event *P.SendStatus* was the *timeout* predicate. In the first step of the verification a global condition of the protocol state is specified in which the *SendStatus* event can be safely executed. This condition is free from real-time aspects, but it is global in the sense that it may involve state variables that non-local to process P. Therefore this condition could not be used in a protocol implementation, but it is useful during the verification. The desired properties are proven using this global condition in the specification.

Then in the second step of the verification, the rate at which the timeout can occur is deduced from the global condition. The rate r of sending state messages will depend on the maximum packet lifetime L in the channels and other protocol parameters.

Global condition

When designing the global enabling condition of *P.SendStatus*, we have to understand the crucial role of duplicate messages in this protocol. It was already mentioned during the description of the channels $C_{p,q}$ and $C_{q,p}$ which model the network layer service that no duplicates are allowed because they can cause protocol errors. The enabling condition of *SendStatus* has to be designed such that no duplicate messages are generated at the transport protocol level, either.

Let D' denote a duplicate of the data message D. Assume that D and D' are received by P at time t_1 and t_2, respectively. If all the data units in the region $[P.pr \ldots D.i -_n 1]$ and the data unit $D.i +_n 1$ were received before t_2, then by executing a sufficient number of *P.AdvWin* and *P.DeliverData* events, *P.pr* and *P.pt* can be advanced such that $D'.i$ falls in the 'can-receive' region again and $P.rcvd[D'.i]$ is `false`.

Therefore D' would be erroneously interpreted by the receiver as the data unit in the next window with sequence number $D.i$. To avoid this hazard, data messages must not be duplicated in $C_{q,p}$ and the sender Q can retransmit a data message only if the previous copy of the message disappeared from the channel. Since the sender counts the periodic state messages to trigger timeouts, the number of state messages in the channel can be used in the global condition which has to assure that the above requirement is satisfied.

The following two conditions must hold when sending a state message [GNS95]:

1. The number of state messages in the channel $C_{p,q}$ never exceeds $m - 1$.
2. If process P sends a state message which when received by Q causes a data message D to be resent by Q, then the previous copy of that data message D' is no longer in the channel $C_{q,p}$.

It is easy to see that the violation of these conditions can lead to duplicate data messages, which then can result in protocol errors as discussed above. It is also apparent that state messages cannot be duplicated either by the channel $C_{p,q}$, because duplicates in the channel could invalidate our assumptions about the number of state messages in transit.

The two conditions above are guaranteed by the following predicate which becomes the enabling condition of the *P.SendStatus* during the first step of the verification:

$$\begin{aligned}&(\#S : S \in C_{p,q}) < m - 1 \wedge (\forall k \in [0 \ldots n-1] : \\ &\quad (\neg P.rcvd[k] \wedge Q.count[k] + (\#S : S \in C_{p,q}) = m - 1) \Rightarrow \\ &\qquad \Rightarrow (\#D : D \in C_{q,p} \wedge D.i = k) = 0)\end{aligned} \tag{3.64}$$

Remember that the notation $(\#P : e)$ was defined in Section 3.2 for the number of packets of type P *ever* sent and which satisfy the predicate e. Thus $(\#D : D \in C_{q,p})$ means the number of data packets *currently* in channel $C_{q,p}$.

Protocol properties

The following invariants were proven in [GNS95]:

$$\Box(Q.qs \in_n [Q.qr, Q.qt) \wedge Q.qs \in_n [P.pr, P.pt) \wedge P.pr \in_n [Q.qr, Q.qs +_n 1) \wedge \\ \wedge \quad Q.qt \in_n [Q.qs +_n 1, P.pt +_n 1) \wedge P.pk \geq Q.qk) \tag{3.65}$$

$$\Box(\forall k \in [0 \ldots n-1] : P.rcvd[k] \Rightarrow k \in_n [P.pr, Q.qs)) \tag{3.66}$$

$$\Box(\forall k \in [0 \ldots n-1] : (\#D : D \in C_{q,p} \wedge D.i = k) \leq 1) \tag{3.67}$$

$$\Box((D \in C_{q,p}) \Rightarrow (\neg P.rcvd[D.i] \wedge D.i \in_n [P.pr, Q.qs))) \tag{3.68}$$

$$\Box((\#S : S \in C_{p,q}) \leq m-1) \tag{3.69}$$

$$\Box((S \in C_{p,q} \wedge S.k > Q.qk) \Rightarrow \\ \Rightarrow \quad (Q.qs \in_n [S.r, S.t) \wedge P.pr \in_n [S.r, Q.qs +_n 1) \wedge P.pk \geq S.k \wedge \\ \wedge (\forall u \in [0 \ldots n-1] : (\neg S.b[u] \wedge u \in_n [S.r, S.t) \wedge \\ \wedge Q.count[u] + (\#S' : S' \in C_{p,q} \wedge S.k \geq S'.k > Q.qk) = m) \Rightarrow \\ \Rightarrow (\neg P.rcvd[u] \wedge (\#D : D \in C_{q,p} \wedge D.i = u) = 0)))) \tag{3.70}$$

The notation $u \in_n [x, y)$ in these assertions means that u is in the modulo-n interval $[x, y)$, where u, x, y are all modulo-n numbers and the interval can "wrap around." Formally this means that

$$u \in_n [x, y) \equiv (x \neq y) \wedge (u = x \vee u = x +_n 1 \vee \ldots \vee u = y -_n 1)$$

The following progress properties were proven in [GNS95]:

$$P.pr = k \rightsquigarrow P.pr = k +_n 1 \tag{3.71}$$

$$Q.qs = k \rightsquigarrow Q.qs = k +_n 1 \tag{3.72}$$

$$P.pt = k \rightsquigarrow P.pt = k +_n 1 \tag{3.73}$$

Note that neither the safety nor the progress properties contain assertions resembling to the desired properties formulated in Section 3.1. Actually, no such properties are

mentioned at all in the paper [GNS95]. If we want to compare this verification to the sliding-window protocol verification of Shankar [Sha89] which was discussed in Section 3.2 or to our verification of PAWS in Section 3.3, then these assertions correspond to the intermediate assertions generated in those proofs. The validity of the desired properties can thus be established with the help of these assertions. Because these extra steps do not add anything new to the understanding of the protocol, we do not document them here.

Real-time constraints

So far (3.64) was used as the enabling condition of the *P.SendStatus* event. In an implementation, the event will be executed r times per second under the control of a real-time clock in process P. An upper bound of r has to be found in such a way that (3.64) is guaranteed to hold whenever the event is triggered based on the real-time clock.

The first conjunct of (3.64) requires that the number of state messages in the channel $C_{p,q}$ never exceeds $m-1$. Since state messages are generated at rate r and they stay at most L seconds in the channel, $r \cdot L$ provides an upper bound on the number of state messages in transit.

At the moment when a data message D is sent, $Q.count[D.i] = 0$ and there are at most $r \cdot L$ state messages in $C_{p,q}$. Just before D reaches P, the condition $Q.count[D.i] + (\#S : S \in C_{p,q}) < m-1$ must hold. The message D takes at most L seconds to reach the receiver P. During this period, $r \cdot L$ more state messages can be generated.

From these considerations the following bound can be deduced for r:

$$r \leq \frac{m-1}{2L} \tag{3.74}$$

Therefore the rate of sending state messages depends on the maximum packet lifetime L and on the number m of state messages that trigger a retransmission.

3.4.3 Eliminating protocol resets

Recall that there are some unbounded non-auxiliary variables in the protocol specification. These are $P.pk$, $Q.qk$ and corresponding field in the state messages $S.k$. Although the formal model assumes that they are unbounded, in an implementation a finite representation must be used. The authors of [GNS95] propose to use a modulo-N representation of these variables and to reset the protocol whenever $P.pk$ is about to wrap around. By selecting a sufficiently large N, the frequency of such protocol resets can be kept arbitrarily low. Note that N is different from the size of the buffer n. If, for example, $N = 2^{32}$ and state messages are generated in every millisecond, then the time between protocol resets would be more than a month.

```
process Q;
    var ...
        qk_o: bool;
        t_qk: epoch;
    event RecvStatus;
            ...
            if S.qk_o ∨ S.k > qk then
                ackd, qr, qt, qk, qr_u := S.b, S.r, S.t, S.k, S.r_u;
                qk_o, t_qk := false, τ;
                ...
    event ExpireQk;
        when timeout do                                   { τ = t_qk + U }
            qk_o := true;
```

Figure 3.16: *Modifications to the specification of the sender P*

Despite this low frequency of resets we propose a simple modification to eliminate resets for the following reasons:

- During a reset no communication is allowed for a period of the maximum packet lifetime. There is no need to suspend the data transfer when our modification is applied.
- Since the cost of resets is eliminated, the value of N can be lowered which saves some header space in messages. As we will see, the minimal value of N depends on the maximum packet lifetime.
- Eliminating the resets simplifies the implementations of the protocol. Such rarely executed actions are usual sources of errors in protocol implementations.

Our proposed modification is based on the perception that the sequence number of state messages is similar by nature to the timestamps in the PAWS specification in Section 3.3. *P.pk* can considered as the equivalent of the clock in the PAWS specification, and $Q.qk$ corresponds to the most recent timestamp. In PAWS, modulo-N timestamps are handled by invalidating the most recent timestamp when it becomes too old. The same can be done with the SNR specification.

The necessary modifications are shown in Figure 3.16. Only the sender Q has to be changed. Two state variables are added: qk_o is a non-auxiliary boolean variable which is set to `true` when qk becomes old; the epoch variable t_{qk} records the time when qk was last refreshed from a state message.

The validation of state messages in event *RecvStatus* is slightly modified. A message S is acceptable if either qk has expired or $S.k > qk$. When a message is accepted, qk_o is set to `false` and the current time is recorded in t_{qk}. If no new state messages are accepted for a time period of U, then the new event *ExpireQk* is triggered and qk_o is set to `true`. The value of the protocol parameter U is determined below.

We want to prove two assertions:

$$\Box(S \in C_{p,q} \wedge Q.qk\_o \Rightarrow S.k > Q.qk) \tag{3.75}$$

$$\begin{aligned}&(\exists K < N : \Box((S \in C_{p,q} \wedge \neg Q.qk\_o) \Rightarrow \\ &\quad \Rightarrow \quad (Q.qk + K \geq S.k \geq Q.qk + K - N + 1)))\end{aligned} \tag{3.76}$$

The assertion (3.75) assures that our modification does not affect the correctness of the protocol. Assertion (3.76) is the correct interpretation condition and its validity assures that $S.k > Q.qk$ in event $Q.RecvStatus$ can be replaced by its modulo-N equivalent, $K \geq S.k -_N Q.qk > 0$.

The proof that (3.75) and (3.76) are valid is relatively simple and its structure is very similar to the corresponding proofs in Section 3.3. Therefore only the outline of this proof is presented here.

Assume that the first state message $S_0 : S_0.i = 0$ is generated at t_0. Because the subsequent messages are generated at a fixed rate r, the message $S_j : S_j.i = j$ is generated at time $t_0 + j/r$. The maximum packet lifetime is L, therefore if S_j is ever received it can only be received in the $[t_0 + j/r, t_0 + j/r + L]$ interval. Finally, we know that $Q.qk\_o$ is `false` if $Q.qk = j$ and S_j was received less than U seconds ago, and $Q.qk\_o$ is `true` if S_j was received more than U seconds ago. Therefore, we can formulate the following invariants:

$$S_j \in C_{p,q} \Rightarrow \tau \in [t_0 + j/r, t_0 + j/r + L]$$

$$Q.qk = j \wedge \neg Q.qk\_o \Rightarrow \tau \in [t_0 + j/r, t_0 + j/r + L + U]$$

$$Q.qk = j \wedge Q.qk\_o \Rightarrow \tau > t_0 + j/r + U$$

From the above invariants the following two invariants can be obtained by transformations:

$$S \in C_{p,q} \wedge Q.qk\_o \Rightarrow S.k \geq Q.qk + \lceil r(U - L) \rceil$$

$$S \in C_{p,q} \wedge \neg Q.qk\_o \Rightarrow Q.qk + \lfloor r(L + U) \rfloor \geq S.k \geq Q.qk - \lfloor rL \rfloor$$

From these we can deduce two conditions for the invariance of (3.75) and (3.76), respectively:

$$L < U \tag{3.77}$$

$$r(2L + U) < N \tag{3.78}$$

The constant K in assertion (3.76) is equal to $\lfloor r(L + U) \rfloor$.

To emphasize the usefulness of our modification, let us consider a example. Let L be 120 seconds which is the currently used value when designing Internet protocols. In the Internet there is no mechanism to control packet lifetimes, but 120 sec is considered to be a safe upper bound on packet lifetimes in any part of the network. The parameters $N = 2^{16}$, $U = 240$ sec, and $r = 100$ Hz satisfy the requirements of (3.77) and (3.78).

On the other hand, if the original specification were implemented using 16-bit acknowledgment sequence numbers, then the protocol would have been reset in every 327 seconds ($= N/2r$). This would mean a 240 sec ($= 2L$) idle period in every 327 sec which is clearly unacceptable.

3.4.4 Alternative specification of SNR

The sliding-window protocol *SW* in Section 3.2 and the protocol *SNR* discussed in this section have a number of similarities. The concept of window is present in both protocols in some form. The 'can-receive' and 'cannot-receive' regions of *SNR* are equivalent with the notion of 'inside the window' and 'out of the window' in the *SW* specification. Both use finite identifiers to refer to the data units transmitted from sender to receiver.

There are differences in the format of messages, the processing rules, and the assumptions about the network service. In our view, the most significant difference is in the relation of two protocol parameters: the size of the sequence number space and the maximum window size. There are separate constants for these values in the specification *SW*: N and RW, respectively. On the other hand, in the specification *SNR* these two values are not independent. The protocol constant n denotes the size of the receive buffer, but the maximum window size is only implicitly defined by $\max(P.pt -_n P.pr)$. Assertion (3.65) implies that this is equal to $n - 1$.

In this section an alternative specification is presented for a data transfer protocol using the idea of periodic state exchange. The specification, called *SNR_2*, is much like the original specification *SNR* except that it decouples the size of the sequence number space n and the maximum window size w. The effects of this modification will be analyzed in Section 3.5 where the properties of the different protocols are compared.

The specification to be presented here can be looked upon in another way. It can be seen as a specialization of the generic sliding-window protocol specification *SW*. It was mentioned in Section 3.2 that any sort of retransmission policy can be used in the implementations of that protocol as long as the fairness assumptions are satisfied. This specification is thus a specific sliding-window protocol which sends acknowledgments, or state messages in the terminology of *SNR*, at regular intervals and counts these acks at the sender to trigger retransmissions.

program *SNR_2*;
type *dataunit* = "the type of data units";
dt = **record of**
i: $0 \ldots n-1$;
data: *dataunit*;
st = **record of**
b: **array** $[0 \ldots w-1]$ **of bool**;
r, t: $0 \ldots n-1$;

channel $C_{p,q}$, $C_{q,p}$;

process P;
var *rcvd*: **array** $[0 \ldots w-1]$ **of bool**;
pr, pt: $0 \ldots n-1$;
See Figure 3.18 for the events of process P.

process Q;
var *src*: **array** $[0 \ldots w-1]$ **of** *dataunit*;
ackd: **array** $[0 \ldots w-1]$ **of bool**;
count: **array** $[0 \ldots w-1]$ **of** $0 \ldots m-1$;
qr, qs, qt: $0 \ldots n-1$;
See Figure 3.19 for the events of process Q.

init $P.pr = Q.qr = Q.qs = 0 \wedge P.pt = Q.qt = 1 \wedge$
$(\forall i \in [0 \ldots w-1] : \neg P.rcvd[i])$;

Figure 3.17: *Alternative SNR specification: main part*

Specification

The alternative specification of the SNR protocol, called *SNR_2* is shown in Figure 3.17, 3.18, and 3.19. To make it simple, all references to the auxiliary variables were removed. Therefore this specification contains only those processing steps that are present in an actual implementation.

The structure of the program *SNR_2* is almost identical to *SNR*. The major difference is that the size of the buffers is w, while the sequence numbers and pointers have a modulo-n representation. To maintain the simple mapping from sequence numbers to buffer indices, $n = kw$ must hold where $k \geq 2$.

Another important change is the lack of the sequence-number field $S.k$ is state messages. State messages do not need their own sequence numbers in this specification because the window information carried in these messages is sufficient to decide whether the message

```
event SendStatus;                                                    { *1* }
    var S: st;
    when timeout do
        S.b, S.r, S.t := rcvd, pr, pt; send(C_{p,q}, S);
event RecvData;                                                      { *2* }
    var D: dt;
    when head(C_{q,p}) ≠ nil do
        D := recv(C_{q,p});
        if D.i ∈_n [pr, pt) then
            rcvd[D.i mod w] := true;
event AdvWin;                                                        { *3* }
    when rcvd[pr mod w] ∧ pt −_n pr > 0 do
        rcvd[pr mod w] := false; pr := pr +_n 1;
event DeliverData;                                                   { *4* }
    when pt −_n pr < w − 1 do
        pt := pt +_n 1;
```

Figure 3.18: *Alternative SNR specification: receiver events*

is new. Each state message of *SNR_2* is equivalent to a combination of a cumulative and some selective acks of SW. The mapping to the cumulative ack CA is as follows:

$$CA.Ack = S.pr, \quad CA.Wnd = S.pt -_n S.pr$$

The selective acks can be formulated from the contiguous ranges of $S.rcvd$.

Most of the events are self explanatory, but some important modifications are highlighted below. In the event *P.RecvData*, the sequence number of the data message D must be within the receive window for the message to be accepted. In the original specification, there was no need for such a check because any data message in the channel was guaranteed to be within the receive window (see assertion (3.68)).

The sender validates state messages in the event *Q.RecvStatus* by looking at the window defined by $S.r$ and $S.t$. If the window has advanced, then S is guaranteed to be new. If the window has not advanced ($Q.qr = S.r \wedge Q.qt = S.t$), then S can be either new or old. Since S may still carry new acknowledgment information for out-of-order data units in $S.b$, the two boolean maps are OR-ed in this case. This assures that no acknowledgment information is lost.

Correctness

The safety of *SNR_2* can be deduced from the safety of SW if we can prove that every computation of *SNR_2* can be mapped to a computation of SW. This is the formal

```
event RecvStatus;                                                    { *5* }
    var S: st;
        j : 0 . . . n − 1;
    when head(C_{p,q}) ≠ nil do
        S := recv(C_{p,q});
        if S.r ∈_n [qr, qs +_n 1) ∧ qt −_n qr ≤ S.t −_n S.r then
            if S.r ≠ qr ∨ qt −_n qr < S.t −_n S.r then
                ackd, qr, qt := S.b, S.r, S.t;
            else
                ackd := ackd ∨ S.b;
            for j = qr . . . qs −_n 1 do
                count[j mod w] := count[j mod w] +_m 1;
                if count[j mod w] = 0 ∧ ¬ackd[j mod w] then
                    D.i, D.data := j, src[j mod w]; send(C_{q,p}, D);
event SendData(d : dataunit);                                        { *6* }
    var D: dt;
    when qt −_n qs > 0 do
        src[qs mod w] := d;
        D.i, D.data := qs, src[qs mod w]; send(C_{q,p}, D);
        count[qs mod w] := 0; qs := qs +_n 1;
```

Figure 3.19: *Alternative SNR specification: sender events*

equivalent of our claim that the specification *SNR_2* is a specialization of *SW* which uses a more restricted retransmission and acknowledgment policy. An alternative way of verifying *SNR_2* would be to carry out the same proof that was used to verify *SW*.

Although we do not present a proof that *SNR_2* implements *SW* in the formal sense, the mapping of an arbitrary computation of *SNR_2* to a computation of *SW* is described informally. A *P.SendStatus* event of *SNR_2* maps to the *R.SendACK* and *R.SendSACK* events of *SW* as it was discussed above.

The *P.RecvData* event corresponds to an *R.RecD* event in *SW*. In this case, however, the mapping is not exact because in *R.RecD* the left edge of the window *R.Nxt* is always advanced if it is possible. The equivalent processing in *SNR_2* is made in the event *P.AdvWin*. Therefore, for *SNR_2* to implement *SW* in the strict sense, any execution of event *P.RecvData* should be followed by as many executions of *P.AdvWin* as possible. The result of this incompleteness in the mapping does not have serious consequences for the correctness of *SNR_2*. The so-called non-interference property, in particular assertion (3.12), does not hold for *SNR_2*, but the desired safety properties do not depend on this assertion.

The event *P.DeliverData* in *SNR_2* corresponds to *R.ExpandWindow* in *SW*. The sender event *Q.RecvStatus* maps to the series of the corresponding *S.RecACK* and *S.RecSACK* events in *SW*, optionally followed by an *S.SendD* event if the reception of the state

message in *SNR_2* triggers a retransmission. The event *Q.SendData* maps to the event *S.Accept* immediately followed by an *S.SendD*.

This completes the informal argument about the safety of *SNR_2*. Because of its relation to *SW*, the same real-time constraints must be satisfied by the protocol:

$$n \geq 2w + L \cdot B \tag{3.79}$$

$$n \geq 2w \tag{3.80}$$

Assertion (3.79), which is the equivalent of (3.24), must hold if the protocol operates over transport channels which can drop, duplicate and reorder packets. The constant B is the maximum transmission rate in data units per second. Assertion (3.80) must hold if the protocol operates over data-link channels which can only drop and duplicate packets, but cannot reorder them.

The progress of *SNR_2* can be proven based on the progress proof of *SW* or on the proof presented in [GNS95]. Note that *SNR_2* implements a receiver-driven retransmission strategy, so it satisfies the third set of liveness assumptions in Section 3.2.3.

3.5 Comparison of protocol variants

Several protocols for reliable data transfer have been analyzed in this chapter so far. To conclude the chapter, we compare them using the results of their verification. This will give us insight to the strengths and weaknesses of each protocol in different environments.

The following protocols will be considered:

- The plain sliding window protocol which is defined by the specification *SW* in Section 3.2.
- The sliding window protocol extended by timestamps, PAWS which is defined by the specification *TS* in Section 3.3.
- Two versions of the periodic state-exchange protocol SNR from Section 3.4 will be used in the comparison. One is the specification *SNR* presented in Section 3.4.1 including the modification proposed in Section 3.4.3. The other is the specification *SNR_2* from Section 3.4.4.

The major comparison criterion is the upper limit on the transmission speed imposed by the correctness conditions of these protocols. We will see that such a bound exists for all the protocols discussed here. The importance of determining this limit on the transmission speed is twofold. On one hand, we must take this limit into consideration during protocol design so that we can assure that the protocol will operate correctly in the targeted networking environment.

On the other hand, to achieve the targeted bandwidth limit with a relatively small header is also important because networks today are largely heterogeneous. For example, in the Internet today users of high-speed testbeds can transfer data approaching Gigabits-per-second while users exist with an access limited to a few kilobits per second. The increasing popularity of mobile access to the Internet indicates that this heterogeneity is a fundamental characteristic of the network. Using as few bits in the headers as possible is important from the viewpoint of these low-speed users. The formulae for the maximum allowed transmission rate of these protocols provides us with means to compare the efficiency of these protocols in utilizing their header space. Naturally, protocols with a better header efficiency are preferred because these present low overhead to low-speed users and still allow high transmission rates for high speed users.

Further comparison criteria are the limitations and assumptions made in the protocols. We will see for example that SNR has some limits on its retransmission policy which may reduce its utility in certain types of environments.

3.5.1 Maximum transmission rate

In each verification a set of real-time constraints were obtained which were sufficient constraints for the correctness of the protocol. Most of the real-time constraints, with the exception of SNR, involve the maximum transmission rate B. Furthermore, each of these constraints gives an upper bound on B. That is, those constraints say that the protocol operates correctly if the maximum rate of sending data units is not higher than a certain bound expressed by other protocol parameters.

This is what one would also expect intuitively: The protocols use bounded identifiers and the channels can reorder the packets. The upper bound on B assures that no identifier is reused before earlier packets carrying the same identifier disappear from the network. Also, this is what mandates the existence of an upper bound on packet lifetimes in the channels, denoted by L.

Bandwidth limit for the plain-sliding window protocol

The real-time constraints for the specification SW is (3.24):

$$N \geq 2RW + L \cdot B$$

This constraint is necessary when the protocol operates over transport channels that can drop, duplicate, and reorder packets. The bandwidth limit from this constraint is:

$$B_{SW} < \frac{N - 2RW}{L} \tag{3.81}$$

If the protocol operates over data-link channels that cannot reorder packets, then the correctness constraint is expressed by (3.26):

$$N \geq 2RW$$

Because of no reordering in the channels, the reception of packet P implies that all packets have disappeared from the network that were sent before P. Therefore, in this case the correct interpretation of identifiers can be guaranteed without limiting the transmission rate.

Bandwidth limit for PAWS

The real-time constraints for the specification *TS* are the inequalities (3.60)–(3.63):

$$N_S \geq 3RW + \lceil \Gamma_S \cdot B \rceil$$

$$N_C \geq \left\lceil \frac{L + TW_R \cdot \Gamma_R}{\gamma_S} \right\rceil + \left\lceil \frac{L}{\gamma_S} \right\rceil + 3$$

$$TW_R \geq \left\lceil \frac{L}{\gamma_R} \right\rceil + 1$$

$$TW_S \geq \left\lceil \frac{2L + TW_R \cdot \Gamma_R}{\gamma_S} \right\rceil + 1$$

To make the comparison with the other protocols easier, we can eliminate the clock rate parameters from these inequalities:

$$B_{TS} < \frac{N_S - 3RW}{\frac{3L}{N_C}} \tag{3.82}$$

This is an absolute upper bound which can only be reached when $\Gamma_S = \gamma_S$ and $\Gamma_R = \gamma_R$, i.e., the clocks have no drift. In the calculations we assumed that $N_C \gg 1$.

Bandwidth limit for SNR

Equation (3.74), (3.77), and (3.78) constitute the real-time constraints for *SNR*:

$$r \leq \frac{m-1}{2L}$$

N [bits]	N_S[bits]	N_C[bits]	B_{SW} [bit/s]	B_{TS} [bit/s]
32	16	16	$268. \cdot 10^6$	$22.4 \cdot 10^6$
48	24	24	$17.5 \cdot 10^{12}$	$1.47 \cdot 10^{12}$
64	32	32	$1.15 \cdot 10^{18}$	$96.1 \cdot 10^{15}$

Table 3.1: *The maximum bandwidth of the plain sliding-window protocol and PAWS for different parameter sets. In all cases* $RW = 0.25N_S$ *and* $L = 120$[s].

$$L < U$$

$$r(2L + U) < N$$

These constraints provide no explicit upper bound for B. It is easy to show, however, that there is an implicit bound on the achievable B for any implementations of this protocol. Each state message contains a bitmap with at most $n - 1$ empty slots for new data. Therefore, in the ideal case, $n - 1$ data units can be transfered per state message. From this we get

$$B_{SNR} < (n - 1)r$$

where r denotes the rate of generating state messages. Combining this expression with the real-time constraints listed above, we get the following bound on the transmission rate:

$$B_{SNR} < \frac{(n - 1)N}{3L} \tag{3.83}$$

The modified SNR specification, *SNR_2* is equivalent to the plain sliding-window protocol in this respect, therefore its limit on the maximum transmission rate is expressed by (3.81).

Calculating the bandwidth limit in different settings

Let us examine the protocols SW and TS first and consider the case when $N = N_S \cdot N_C$ because the header space is then equal for the two protocols. It is easy to show that the bound on the bandwidth is tighter in case of TS for any parameter set. Table 3.1 shows the exact values for three different situations. In this particular parameter set, $B_{SW} \approx 12B_{TS}$.

The table shows, for example, that current high speed networks approach the maximum allowed transmission rate of TCP [Pos81a] which uses the plain sliding-window protocol with 32-bit sequence numbers. The bandwidth limit of the 48 and 64-bit versions is

N_S[bits]	N_C[bits]	B_{TS} [bit/s]	γ [ms]
16	16	$4.19 \cdot 10^6$	15.6
24	24	$16.8 \cdot 10^9$	1.00
32	32	$4.29 \cdot 10^{12}$	1.00

Table 3.2: *The maximum bandwidth of PAWS when some practical limitations are also considered. Parameters not shown here are identical to those in Table 3.1.*

in the Terabit/sec range and beyond therefore it will not be a serious limitation in the foreseeable future.

In case of PAWS, the maximum transmission rate that belongs to the 24-bit sequence numbers and 24-bit timestamps case is also over 1 Terabits/s. Note, however, that the values for B_{TS} in the table are optimal values that could only be achieved with specific protocol parameter settings. In most practical situations, a number of additional constraints must be satisfied. Some of these are listed below:

- The correct interpretation conditions determine how comparisons on modulo-N numbers must be implemented (see (3.5 in Section 3.2.2). This involves the test $K \geq a -_N b$, where a and b are the modulo-N numbers to compare and K is the constant from the CI condition. This test is easy to implement using 2-complement arithmetic instructions if $K \leq N/2$.

  In case of PAWS this translates to the additional constraints: $TW_S \leq N_C/2$ and $TW_R \leq N_C/2$. In theory, this argument holds for the plain sliding-window protocol as well, but it has no practical consequences in that case because $RW \leq N/2$ is already implied by the correctness constraints.
- Since most implementations of the *TS* specification will be both senders and receiver, it is convenient to use the same clock rate for the clocks of the sender and receiver. That is, $\Gamma_S = \Gamma_R = \Gamma$ and $\gamma_S = \gamma_R = \gamma$.
- When calculating the bandwidth limit, we assumed no clock drift. In practice one must count on some clock drift. There are two reasons to define $\Gamma > \gamma$ for the protocol. One reasons is that it is impossible to implement infinitely accurate clocks. The other reason is to give system designers more freedom in selecting the appropriate clock rate for their implementations. The range of convenient clock rates may vary from one system to another. That is the major reasons that the PAWS specification defines $\Gamma = 1$sec and $\gamma = 1$msec.

Based on these considerations, we calculate more realistic values of B_{TS}. These are shown in Table 3.2. The following assumptions were used to obtain the values in the table: $\Gamma = 2\gamma$, $\gamma \geq 1$msec, and $TWs \leq N_C/2$. The results for B_{TS} are considerably lower than those in Table 3.1.

From the calculations above we can conclude that *TS* uses the identifier space less efficiently than *SW*. Note, however, that the PAWS mechanism used in *TS* provides extra functionality with respect to the plain sliding-window protocol. The timestamps can be used for simpler and more accurate round-trip time measurements. Therefore, it

is a clear trade-off between extra functionality and the size of the header space used by the protocol.

If we calculated B_{SNR} from (3.83) using the same parameters, then we would get a huge number, orders of magnitude higher than B_{SW}. This comparison would not be realistic, however. In SNR, packets are the units of numbering not octets. A consequence is that the size of the buffer n is expected to be much smaller than the window size RW in the sliding-window protocols. Note also that the size of state messages is linearly increasing with n which places a practical upper limit on n.

A similar comparison can be made, however, between *SNR* and *SNR_2*. Again, the parameters are selected in such a way that the header sizes are roughly equal. Let $n = 2^8$ and $N = 2^{16}$ for the specification *SNR*, and let $w = 2^8$ and $n = 2^{16}$ for *SNR_2*. Substituting these parameters into (3.83) and (3.81), respectively, we see that B_{SNR} is almost two orders of magnitude higher than B_{SNR_2}. As in the case of SW and TS, the higher allowed transmission rate does not come for free. The specification *SNR* has some limitations regarding the retransmission of lost data. This will be examined in the next section.

3.5.2 Limitations on the retransmission policy

Let us consider the situation in *SNR* when a data message is lost. Assuming that no other messages are lost and the delay in the network is constant, state messages arrive at the sender at regular intervals, one in every $1/r$ seconds. The m^{th} state message arrives sometimes in the $[(m-1)/r, m/r]$ interval. It follows from (3.74) that the lower bound of the interval is at least $2L$. Should state messages be lost, the retransmission happens even later. Note also that the lower bound on the retransmission delay does not depend on the parameter m.

In most datagram networks the value of L can be rather large because there is no explicit mechanism to enforce packet lifetimes. Such a network is the current Internet which defines L to be 120 seconds, although the average delay is not likely to exceed a few hundred milliseconds even on intercontinental paths [SAGJ93]. Some argue that future networks will not enforce stricter bounds either because of the high cost of accurate lifetime enforcement [Che89].

Over such networks the protocol specified by *SNR* does not operate efficiently in face of losses. If the sender receives no acknowledgment of a data message within the average round-trip delay, then the message is *likely* to be lost. This protocol, however, must wait with the retransmission until it is *guaranteed* to be lost and that can degrade the performance considerably.

A related issue is that *SNR* does not allow duplicates in the channels. This is also difficult to enforce in many networks. For example, duplicates of up to 1% of all transmitted packets were measured in certain parts of the Internet [SAGJ93]. Connection oriented

networks, such as ATM, do not duplicate packets and the specification *SNR* can be applied in such environments. On the other hand, these networks do not reorder packets either. Therefore their service can be modeled by the so-called data-link channels and in that case *SNR_2* has no bound on B at all.

Chapter 4

Connection Management Protocols

Protocols for the reliable opening of connections are studied in this chapter. As in the previous chapter, the study is based on the formal verification of different connection management (CM) protocols. The results of these verifications are then used to compare the protocols and to discuss their design.

In Section 4.1, our generic model of CM protocols is introduced. The protocol properties which we consider essential for all CM protocols are also introduced here.

Section 4.2 discusses the verification of SCMP [OHdG95b, OHdG95c]. SCMP [LSW91] is a novel protocol for at-most-once message delivery based on the assumption that globally synchronized clocks are cheap to implement. From this basic assumption a very interesting protocol was developed which uses timestamps to identify messages. Because the problem of reliably opening a connection is essentially equivalent to at-most-once message delivery, SCMP can be used as the basis of a 2-way handshake (2WHS) CM protocol.

CM protocols using the 3WHS were extensively studied before, see e.g. [MS87]. Recently, a number of CM protocols have been proposed which fall between the categories of 2WHS and 3WHS. These protocols use the faster 2WHS whenever possible. In cases when there is not sufficient information available in the connection records for reliably opening a connection with the 2WHS, they fall back to using a 3WHS scheme. The expected advantage of such schemes is that the information which must be retained for 2WHS schemes to work can be treated as cached information. If the information is present, then it speeds up the connection setup, but it can be discarded any time without loss of functionality. Section 4.3 is devoted to the analysis of such protocols.

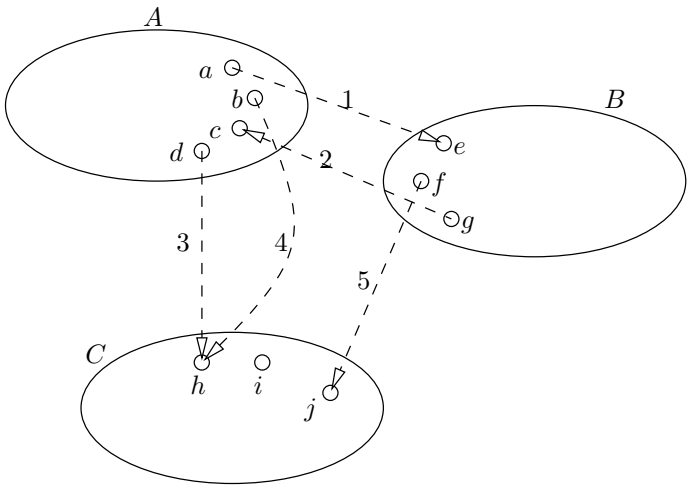


Figure 4.1: *Endpoints, hosts and connections.*

4.1 Desired properties

In our model, the communication takes place between *connection endpoints*. Each endpoint belongs to exactly one *host*, but a host may have several endpoints (Figure 4.1). Hosts model the physical entities (e.g. computers) on which the protocol executes.

Between each pair of endpoints there is a (potential) *connection*. Each connection may become open and closed many times over time. To distinguish among the different open periods of a connection, we refer to *connection incarnations*. Every time a new attempt is made to open the connection, a new incarnation is started.

Figure 4.1 shows 3 hosts (A, B, C); each of them has a number of connection endpoints. Host A, for example, has the endpoints a, b, c, and d. Connections take place between endpoints. There may be several connections between two hosts if they belong to different endpoints, e.g., connections 1 and 2 in the figure. This explicit notion of hosts in our model allows us to describe protocols which have state variables belonging to all the connection endpoints of a certain hosts.

The protocol entities can communicate with each other by sending packets over channels provided by the network-level service. Conceptually, there are two channels between any two endpoints a and b: $C_{a,b}$ from a to b and $C_{b,a}$ in the opposite direction. Usually the network service does not provide channels to individual endpoints—like in the Internet where IP datagrams are delivered between hosts. However, including a source and destination endpoint address in every packet would allow to distinguish between packets on the different logical channels. Therefore, we just simply assume that there is a channel from any endpoint to any other endpoint.

Our model of connection endpoints and hosts is very general. In some protocol verifications the notion of hosts is not used, see e.g., [LLSA93], [SL95]. This can be considered as a special case of our model when each host has exactly one endpoint address. The

```
program CM;
    type
        ep_addr = "type of endpoint addresses";
        conn_rec = record of
            status: {closed, opening, open};
            lin: int;
            open_to: array [0 ... ∞] of int ∪ empty;
    var
        CR: array [ep_addr, ep_addr] of conn_rec;
    process Entity;
        event BeginIncarnation(a, b);
            when CR[a, b].status = closed do
                CR[a, b].status := opening; CR[a, b].lin + +;
        event OpenIncarnation(a, b, rin);
            when CR[a, b].status = opening do
                CR[a, b].status := open;
                CR[a, b].open_to[CR[a, b].lin]] := rin;
        event CloseIncarnation(a, b);
            when CR[a, b].status ≠ closed do CR[a, b].status := closed;
    init
        (∀a, b : CR[a, b].status = closed ∧ CR[a, b].lin = −1∧
            ∧(∀i : CR[a, b].open_to[i] = empty));
```

Figure 4.2: *Abstract connection management service*

reason for introducing the more general model with the notion of hosts is that it makes possible to model protocols where several logical connections share a global variable in a host. Such a global variable is the ISN (initial sequence number) counter in TCP [Dal75, Tom75], or the monotonic clock in SCMP.

Figure 4.2 shows the definition of the abstract CM service based on the definition given in [Sha91]. Our only modification is the introduction of the notion of hosts with respect to the specification in [Sha91].

Information about connections is stored in *connection records* (CR). In the model there is a CR for each end of each possible connection. If we consider the example of Figure 4.1 again, there are two CRs assigned to connection 1: $CR[a, e]$ and $CR[e, a]$. $CR[a, e]$ records the status of the connection from the point of view of endpoint a and $CR[e, a]$ from the point of view of endpoint e. Note, however, that most of these records are needed only in the formal model. In an implementation, only those CRs occupy real memory which are in a status other than closed.

The *status* maintained in a CR refers to the status as observed by the endpoint to which the CR belongs. The value of *lin* gives the current or last incarnation of the connection. The value of $open\_to[i]$ gives the remote incarnation number to which the local

incarnation i was connected. Therefore, $CR[a, e].open\_to[i] = j$ means that incarnation i of endpoint a is or has been connected to incarnation j of endpoint e as far as a is concerned. Later we will see that it is possible that incarnation j of e has not been connected to incarnation i of a.

In the specification, the CR array is defined as a global variable, i.e., syntactically it is not in the scope of any process. This is in contradiction to the distributed system model which was given in Section 2.3 because we said that all state variables were local to processes. Actually, this assumption is not violated here. The reason for defining CR this way is only to simplify the notation.

To see that the locality assumption of state variables still holds, consider the following. There is an instance of the process *Entity* running for each host in the network. The process at host A accesses only those connection records $CR[a, b]$ where endpoint a belongs to host A. Therefore, any $CR[a, b]$ can only be accessed by a single process only (the one which "runs on" host A) and the locality assumption is satisfied.

Normally a connection incarnation cycles through the states of `opening`, `open` and `closed`. An incarnation may also become `closed` without ever being `open` to a remote incarnation. This is the case when a connection attempt fails because of network error or because of the unwillingness of the remote endpoint to engage in a connection. On the other hand, no incarnation may become `open` more than once because *BeginIncarnation* event always starts a new incarnation when it changes the *status* to `opening`.

In order to specify the desired properties of the CM service, we define the meaning of $connected_{[a,b]}(i, j)$:

$$connected_{[a,b]}(i, j) \equiv CR[a, b].open\_to[i] = j \vee CR[b, a].open\_to[j] = i$$

where a and b denote different endpoint addresses, i and j denote non-negative integers. Using this definition, the desired safety property can be expressed by (4.1):

$$\Box((connected_{[a,b]}(i, j) \wedge connected_{[a,b]}(k, l)) \Rightarrow (i = k \Leftrightarrow j = l)) \tag{4.1}$$

The desired property implies that connections form a 1–1 association between the open incarnations of endpoint a and b. Of course, this property is expected to hold for any pair of different endpoints.

4.2 SCMP

SCMP (Synchronized Clock Message Protocol), which is the subject of the analysis in this section, is based on the novel idea of using synchronized clocks [LSW91]. In the description of SCMP, we will make a distinction between clients and servers. Clients are

those endpoints which actively open a connection by sending an open request. Servers are passive; they open a connection on the receipt of a valid open request but do not initiate connections themselves. Each connection endpoint is either a client or a server, but never both. A host, however, can have many endpoints some of which are clients and the others are servers.

The client associates a timestamp with each connection request and the server uses the associated timestamp to decide whether or not to accept a received request. If the server remembers the timestamp *ts* of the last accepted request from the client, then a request is new if its timestamp is greater than *ts*. However, it is not feasible to keep the last timestamp of every client due to the vast number of clients. Therefore, the server may discard state information of idle connections, but it maintains the variable *upper* which is the maximum of the discarded *ts* values.

If the server does not remember the latest timestamp from a client when a request arrives, then the request is accepted if this timestamp is greater than *upper*. Comparing the timestamp in the request to *upper* assures that duplicates are never accepted. On the other hand, a non-duplicate request may be rejected if its timestamp is less than *upper*. The probability of this can be kept sufficiently low due to the roughly synchronized clocks.

We do not model the clock synchronization protocol here. For our analysis it is sufficient that there exist protocols [Mil91] which can synchronize clocks of computers even on a wide-area network to a few hundred milliseconds with very high probability but without an absolute guarantee that the clocks remain always synchronized. The fact of clock synchronization appears in our model in the form of real-time assumptions and we know that they can be implemented by using a clock synchronization protocol.

SCMP was designed in such a way that "if the rare event of unsynchronized clocks does occur, the protocol continues to work correctly, although there may be a degradation of performance" [LSW91]. The main result in this section is that we formally prove the above statement, namely that SCMP satisfies its safety requirements without assuming that the clocks are always synchronized. We consider the case of using unbounded timestamps and the case when timestamps are from a modulo-N space.

In the case of unbounded timestamps, the monotonicity of the clocks is the only requirement for the correct operation of SCMP. When modulo-N timestamps are used, we need to make further assumptions. Specifically, we have to assume that packet delays in the network, the duration of a connection incarnation, and the rate of the clocks are all bounded. These are not heavy assumptions because such assumptions have to be made for every reliable protocol that uses a bounded identifier space over a network which reorders packets [SL93, Wat81]. Notice that the bounded clock rate assumption does not imply synchronized clocks.[1]

Our strategy for verifying SCMP is similar to that used in the previous chapter. We start

[1]The difference between any two clocks should also be bounded for the clocks to be synchronized.

program *SCMP*;
type
ep_addr = "type of endpoint addresses";
clnt_conn_rec = **record of**
status: {`closed`, `opening`, `open`};
ts, *lin*: **int**;
ts_sent: **array** $[0 \ldots \infty]$ **of int** $\cup$ `empty`;
open_to: **array** $[0 \ldots \infty]$ **of int** $\cup$ `empty`;
srvr_conn_rec = **record of**
status: {`closed`, `open`};
ts, *lin*: **int**;
ts_rcvd: **array** $[0 \ldots \infty]$ **of int** $\cup$ `empty`;
open_to: **array** $[0 \ldots \infty]$ **of int** $\cup$ `empty`;
OpenReq = **record of**
ts, *lin*: **int**;
OpenAck = **record of**
ts, *lin*, *rin*: **int**;
channel one channel $C_{a,b}$ for each a, b pair of endpoints;
Client process in Figure 4.4;
Server process in Figure 4.5;
Initial condition in Figure 4.6;

Figure 4.3: *SCMP: main program*

by specifying and verifying the protocol with unbounded identifiers, and then verify the modulo-N protocol based on these results. In this case, however, we will also see how an ad-hoc verification method can lead to different conditions for the correctness of the protocol. The formal proofs are presented in Appendix B and they can also be found in [OHdG95c].

4.2.1 Specification of unbounded SCMP

The specification can be seen in four figures: Figure 4.3 shows the main program, Figure 4.4 and 4.5 show the client and server processes, and Figure 4.6 shows the initial conditions.

State variables

Every host, either client or server, maintains a monotonic clock *time*. Also, every host H has a connection record $CR(a, b)$ for every (a, b) pair, where a and b are different endpoint addresses and the host of a is H.

```
process C;
    var time: int;
        CR: array[ep_addr, ep_addr] of clnt_conn_rec;

    event ClkTick;
        when true do
            time++;
            for each local CR[a, b] do
                if CR[a, b].status ≠ closed ∧ CR[a, b].ts = time − W_C then
                    CR[a, b].status := closed;

    event Open(a, b);
        when CR[a, b].status = closed do
            CR[a, b].status := opening; CR[a, b].ts := time;
            CR[a, b].lin++; CR[a, b].ts_sent[CR[a, b].lin] := time;

    event SendPkt(a, b);
        var R: OpenReq;
        when CR[a, b].status = opening do
            R.ts := CR[a, b].ts; R.lin = CR[a, b].lin);
            send(C_{a,b}, R);

    event RecvPkt(a, b);
        var A: OpenAck;
        when head(C_{b,a}) = A do
            if CR[a, b].status = opening ∧ CR[a, b].ts = A.ts then
                CR[a, b].status := open;
                CR[a, b].open_to[CR[a, b].lin] := A.lin;

    event Close(a, b);
        when CR[a, b].status ≠ closed ∧ CR[a, b].ts < time do
            CR[a, b].status := closed;
```

Figure 4.4: *SCMP: client process*

In the CR maintained by a client, *status* and *ts* are non-auxiliary variables; *lin*, *ts_sent* and *open_to* are auxiliaries. The variables *status*, *lin*, and *open_to* have already been explained when we specified the abstract CM service in Section 4.1. The variable *ts* contains the timestamp assigned to the current connection request when *status* ≠ `closed`. *ts_sent*[*i*] records the timestamp assigned to incarnation *i* or it is equal to `empty` if that incarnation does not exist yet.

In case of a server, the variable *status* can be either `closed` or `open`. The state `opening` is not needed because, as we will see later on, server incarnations enter the `open` state

directly upon the receipt of a valid open request. The variable *ts* holds the timestamp of the most recently accepted request. $ts\_rcvd[i]$ stores the timestamp of the request accepted by incarnation i. This is an auxiliary variable. Apart from the per-connection CRs, each server maintains a non-auxiliary variable *upper* which gives an upper limit of the timestamps in the CRs of the closed connections.

Events

The *time* variable is regularly incremented by the *ClkTick* event at both clients and servers. This event also enforces a maximum connection lifetime; when a connection becomes too old, then the connection is forcibly closed. W_C and W_S are the constants that give the maximum lifetime of a connection as measured on the clock of the client and server, respectively. The server also updates *upper* in the *ClkTick* event. The triggering of the *ClkTick* event is controlled by the clock synchronization protocol, but because this protocol is not modeled here, we simply enable the *ClkTick* events continuously. When necessary, we will use assumptions to bind the advance of the clocks to real time.

The $C.Open(a, b)$ event initiates a new connection incarnation. The status of the connection is changed to `opening` and the timestamp of the request is stored in the CR. While in the `opening` state, the client can (re)send the connection request. The fields of a connection request are the timestamp *ts*, and the client incarnation number *lin*.

When the server receives a request R, it looks up the appropriate CR using the source and destination of the request. If there is status information about this connection ($status \neq$ `closed`), then the request is a new one if $R.ts > S.CR[a, b].ts$. Otherwise, the timestamp of the request is compared to *upper*.

Messages that are 'newer' than $time + \epsilon$ are not accepted because they are too early. These early messages are indicators of synchronization problems and accepting them could lead to the blocking of requests from well synchronized hosts. Another reason to reject early requests is that the crash recovery mechanism is also based on the assumption that a known upper limit of the timestamps that could be ever accepted exists. ϵ is a specification parameter which gives the maximum expected difference of the clocks. If the request is acceptable, the server starts a new connection incarnation and enters the `open` state.

While in the `open` state, the server sends acknowledgments to the client. An acknowledgment contains a timestamp and two auxiliary fields, the local and remote incarnation numbers. The client checks an incoming acknowledgment by comparing its timestamp to the timestamp stored in the CR. If the state of the connection is `opening` and the timestamps are equal, then the acknowledgment is accepted and the state is changed to `open`.

At the client, the state of a connection can only be changed to `closed` when the clock has advanced at least one tick since the connection was started. This is to prevent reusing

```
process S;
    var time, upper: int;
        CR: array [ep_addr, ep_addr] of srvr_conn_rec;

    event ClkTick;
        when true do
            time++;
            for each local CR[a,b] do
                if CR[a,b].status ≠ closed ∧ CR[a,b].ts = time − W_S then
                    CR(a,b).status := closed;
            upper := max(upper, time − W_S);

    event RecvPkt(a,b);
        var R: OpenReq;
            l: int;
        when head(C_{b,a}) = R do
            if CR[a,b].status = closed then
                l := upper;
            else if CR[a,b].status ≠ closed then
                l := CR[a,b].ts;
            if l < R.ts ≤ time + ε then
                CR[a,b].status := open; CR[a,b].ts := R.ts;
                CR[a,b].lin++;
                CR[a,b].ts_rcvd[CR[a,b].lin] := R.ts;
                CR[a,b].open_to[CR[a,b].lin] := R.lin;

    event SendPkt(a,b);
        var A: OpenAck;
        when CR[a,b].status = open do
            A.ts := CR[a,b].ts; A.lin := CR[a,b].lin;
            A.rin := CR[a,b].open_to[CR[a,b].lin];
            send(C_{a,b}, A);

    event Close(a,b);
        when CR[a,b].status = open do
            upper := max(upper, CR[a,b].ts);
            CR[a,b].status := closed;
```

Figure 4.5: *SCMP: server process*

init

$C.time = S.time = 0 \wedge$
$(\forall a, b : a \text{ is client, } b \text{ is server :}$
$C.CR[a,b].status = S.CR[b,a].status = \texttt{closed} \wedge$
$C.CR[a,b].lin = S.CR[b,a].lin = -1 \wedge$
$C.CR[a,b].ts = S.CR[b,a].ts = 0 \wedge$
$(\forall i :$
$C.CR[a,b].open\_to[i] = S.CR[b,a].open\_to[i] = \texttt{empty} \wedge$
$C.CR[a,b].ts\_sent[i] = S.CR[b,a].ts\_rcvd[i] = \texttt{empty}));$

Figure 4.6: *SCMP: initial condition*

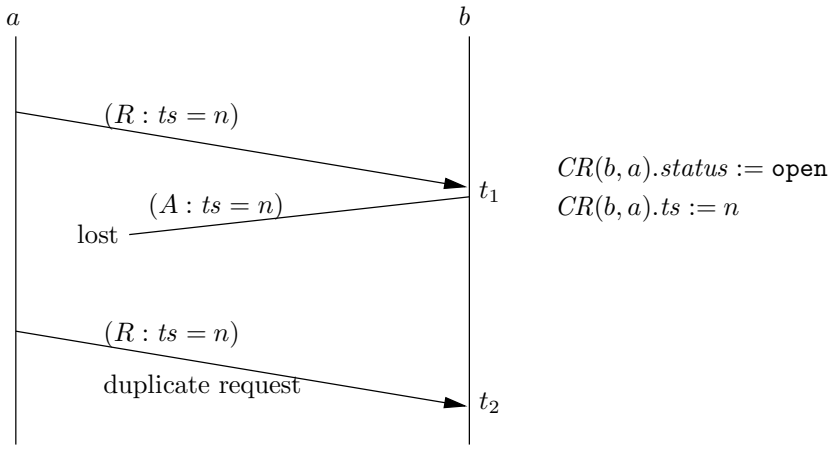


Figure 4.7: *Duplicate request at the server.*

the same timestamp in different connection incarnations.

A scenario when server b receives a duplicate request from client a is shown in Figure 4.7. The first copy of the packet is received at time t_1. The request is accepted, therefore *status* becomes `open` and the timestamp n of the request is saved in ts. The duplicate is received at t_2. There are two cases to consider depending on the state of the server at t_2:

- If the state is still `open` then the timestamp n in the packet is compared to $CR[b,a].ts$ which is also equal to n, thus the request is not accepted again[2].
- If the state is `closed`, then there must have been a $Close(a,b)$ event between t_1 and t_2. The *Close* event assures that *upper* is not lower than n after its occurrence. Therefore the duplicate request is rejected because its timestamp n is less than or equal to *upper*.

The duplicate is detected by the server because it has either the timestamp (when

[2]The reception of a duplicate request may trigger the retransmission of the acknowledgment in an implementation because the duplicate usually indicates that the previous acknowledgment was lost. Note this is allowed by our specification which allows almost any sort of retransmission strategy.

status = `open`) or an upper limit of the timestamp (when *status* = `closed`) of the most recent request from the client.

The monotonicity of the clocks is crucial for the detection of duplicates which will become more evident during the verification. Additionally, the synchronization of the clocks is needed to have a good estimate of the acceptable timestamps in *upper* and in the check for too early requests in *S.RecvPkt*. Therefore the synchronization is only needed to assure that valid requests are not rejected.

4.2.2 Desired safety properties of SCMP

Our goal is to show that SCMP satisfies the safety requirement (4.1) of the abstract CM service. Instead of proving this directly, we show that SCMP satisfies some stronger assertions (4.2), (4.3) and (4.4).

These assertions are expected to hold for all different a, b endpoint addresses and for all i, j, k integers[3]:

$$S.CR[a,b].open\_to[i] = j \rightarrow C.CR[b,a].open\_to[j] \in \{i, \texttt{empty}\} \tag{4.2}$$

$$C.CR[a,b].open\_to[i] = j \rightarrow S.CR[b,a].open\_to[j] = i \tag{4.3}$$

$$S.CR[a,b].open\_to[i] = j \wedge k \neq i \rightarrow S.CR[a,b].open\_to[k] \neq j \tag{4.4}$$

Assertion (4.2) requires that once the server incarnation i becomes open to the client incarnation j, then at that and any later time j is either open to i or j has not been open to any other incarnation. Assertion (4.3) is a similar statement, but from the client's viewpoint. If the client incarnation i is open to the server incarnation j, then at that and any later time j must be open to i. Assertion (4.4) requires that if the server incarnation i is open to the client incarnation j, then there may be no other server incarnation k which is also open to j.

Lemma 4.1 *The assertions (4.2), (4.3) and (4.4) imply the truth of assertion (4.1).*

Although these assertions are stronger than the minimal requirements, we expect that it is easier to prove that they are satisfied by SCMP because they reflect the way SCMP works. For example, (4.1) does not say anything about the order in which a pair of endpoints become open to each other. In case of SCMP, however, always the server becomes open to the client first which is reflected by (4.2) and (4.3).

[3]Please note that the constants i, j and k represent *non-empty* values, therefore the $x = j$ proposition implies that $x \neq$ `empty`.

4.2.3 Proving the safety of unbounded SCMP

Theorem 4.2 *SCMP satisfies the desired safety properties (4.2)–(4.4).*

The main steps of the proving Theorem 4.2 are shown here by listing the assertions we used to prove the invariance of the desired properties.

The first group of assertions provides the link between timestamps and incarnation numbers. (4.5) states that if there is a request R in the channel, then its timestamp $R.ts$ is equal to the timestamp of the client incarnation $R.lin$ which is stored in the history variable ts_sent of the corresponding CR. (4.6) says that if server incarnation i is connected to the client incarnation j, then the timestamp $ts\_rcvd[i]$ received by incarnation i is equal to the timestamp $ts\_sent[j]$ sent by incarnation j. (4.7) is similar to (4.2), but it is for acknowledgments: if there is an ack in the network then its rin and ts fields are equal to the appropriate history variables of the server.

$$R \in C_{a,b} \rightarrow R.ts = C.CR[a,b].ts\_sent[R.lin] \neq \texttt{empty} \tag{4.5}$$

$$\begin{aligned} &S.CR[a,b].open\_to[i] = j \rightarrow \\ &\quad \rightarrow \quad S.CR[a,b].ts\_rcvd[i] = C.CR[b,a].ts\_sent[j] \neq \texttt{empty} \end{aligned} \tag{4.6}$$

$$\begin{aligned} &A \in C_{a,b} \rightarrow (A.rin = S.CR[a,b].open\_to[A.lin] \neq \texttt{empty} \wedge \\ &\quad \wedge \quad A.ts = S.CR[a,b].ts\_rcvd[A.lin] \neq \texttt{empty}) \end{aligned} \tag{4.7}$$

(4.6) and (4.6) are preconditions of the required properties (4.2) and (4.3) with respect to the $C.RecvPkt$ event of the client. (4.6) is also a precondition of (4.7) with respect to the event $S.SendPkt$; (4.5) is the precondition of (4.6) with respect to $S.RecvPkt$.

The following two assertions (4.8) and (4.9) state that the timestamps of successive incarnations are monotone increasing. (4.8) is a precondition of (4.3) with respect to $C.RecvPkt$. (4.9) is a precondition of (4.4) with respect to $S.RecvPkt$.

$$\Box(0 \leq i < C.CR[a,b].lin \Rightarrow C.CR[a,b].ts\_sent[i] < C.CR[a,b].ts\_sent[i+1]) \tag{4.8}$$

$$\Box(0 \leq i < S.CR[a,b].lin \Rightarrow S.CR[a,b].ts\_rcvd[i] < S.CR[a,b].ts\_rcvd[i+1]) \tag{4.9}$$

Finally, (4.10), (4.11), (4.11) and (4.13) are assertions that are sufficient preconditions for the preservation of (4.8) and (4.9). (4.11) states that if a connection is closed as

observed by the client, then the clock of the client is greater than the timestamp of the most recent incarnation. This is a precondition of (4.8) with respect to $C.Open$. (4.12) and (4.13) are the preconditions of (4.9) with respect to $S.RecvPkt$.

$$\Box(C.CR[a,b].status \neq \texttt{closed} \Rightarrow \\ \Rightarrow \quad C.CR[a,b].ts\_sent[C.CR[a,b].lin] = C.CR[a,b].ts) \tag{4.10}$$

$$\Box(C.CR[a,b].status = \texttt{closed} \Rightarrow \\ \Rightarrow \quad C.CR[a,b].ts\_sent[C.CR[a,b].lin] < C.time) \tag{4.11}$$

$$\Box(S.CR[a,b].status \neq \texttt{closed} \Rightarrow \\ \Rightarrow \quad S.CR[a,b].ts\_rcvd[S.CR[a,b].lin] = S.CR[a,b].ts) \tag{4.12}$$

$$\Box(S.CR[a,b].status = \texttt{closed} \Rightarrow \\ \Rightarrow \quad S.CR[a,b].ts\_rcvd[S.CR[a,b].lin] \leq S.upper) \tag{4.13}$$

Extending (4.2)–(4.13) with some trivial assertions, one can prove that the safety requirements (4.2), (4.3) and (4.4) are indeed invariants of unbounded SCMP. Because during these proofs we made no assumptions about the difference of the clocks, we can conclude that unbounded SCMP satisfies its safety requirements without the need for the clocks being synchronized. The safety of the protocol depends only on the monotonicity of the clocks which is reflected by assertions (4.8) and (4.9) among others.

Another consequence of having unbounded timestamps is that there are no bounds on the value of the protocol parameters W_C and W_S. In fact, the verification remains valid even if these parameters approach infinity. In other words this means that there is no limit on the incarnation lifetime if the timestamps are unbounded.

4.2.4 SCMP with modulo-N timestamps

The second part of our analysis is concerned with the properties of SCMP when timestamps are from a finite space. We are interested in the extra constraints, if any, that are necessary for the safety of the modulo-N protocol.

The use of modulo-N timestamps in SCMP means that the same modulo-N timestamp is assigned to different incarnations over time. It is clear that a timestamp can only

be reused when the packets carrying the same timestamp from earlier incarnations have disappeared from the network. To allow the reuse of the timestamps, we will assume that the network does not delay packets longer than L seconds.

Furthermore, we also have to assume that neither the client, nor the server keep a timestamp in use indefinitely. The server must not keep a connection open too long, otherwise a new incarnation started by the client using the same timestamp could misinterpret an old acknowledgment as an acknowledgment of the new incarnation. The client must also limit the lifetime of an incarnation in order to allow the server to discard the CR of old connections. The constants W_C and W_S in the specification give the maximum lifetime of a client and server incarnation, respectively.

W_C and W_S are given as the number of ticks of the local clock. In order to be useful for limiting the lifetime of an incarnation, the rates of the clocks have to be bound to the real time. Therefore, we will assume that the rate of any clock is within certain bounds.

Two different techniques will be used to verify the modulo-N protocol. One is based on the correct interpretation (CI) conditions which was used in the previous chapter. With this technique we get that clock synchronization is sufficient for the safety of the protocol. Clock synchronization means the existence of an upper bound on the difference of any two clocks in the system. Note that this condition is stronger than the initial assumption which implied only probabilistic clock synchronization.

By using an alternative method we will be able to show that clock synchronization is not necessary for the safety; limited packet lifetime and bounded clock skew are also sufficient conditions for the safety of modulo-N SCMP.

4.2.5 Correct interpretation conditions

When using this method, a CI condition is formulated for every comparison on modulo-N variables that occurs in the protocol. The modulo-N version of the protocol is obtained by replacing the unbounded non-auxiliary variables by modulo-N variables. In case of SCMP this affects the *time* and *ts* variables of the client and the sender, and the *ts* header field in the packets. The auxiliary variables may remain unbounded, because their value does not influence the behavior of the system.

The following CI conditions can be formulated for SCMP. In the event $C.ClkTick$, $C.CR[a,b].ts$ is compared to $C.time - W_C$. These two variables are also compared in $C.Close$. Due to the monotonicity of $C.time$ and because $C.ClkTick$ automatically closes incarnations whose timestamp becomes too old, the assertion

$$\begin{aligned}&\Box(C.CR[a,b].status \neq \texttt{closed} \Rightarrow \\ &\quad \Rightarrow \quad C.CR[a,b].time \geq C.CR[a,b].ts > C.CR[a,b].time - W_C)\end{aligned} \tag{4.14}$$

is satisfied by the system. Assertion (4.14) implies the CI requirement obtained by

substituting 0 for K in (3.5) provided that

$$W_C \leq N \tag{4.15}$$

Note also that the comparisons are only effective when $C.CR[a,b].status \neq \texttt{closed}$. Including this in the CI condition (4.14) makes it somewhat weaker. Because the CI conditions are *sufficient* requirements for the safety of modulo-N SCMP, we would like to make them as weak as possible in order to obtain the sharpest conditions for safety.

The client also does a comparison in *C.RecvPkt* to check the timestamp of the acknowledgment received. Again, we use $K = 0$ in the CI formula (3.5), because we expect that

$$\Box(A \in C_{b,a} \wedge C.CR[a,b].status \neq \texttt{closed} \Rightarrow C.CR[a,b].ts \geq A.ts)$$

holds. Using this the CI requirement becomes

$$\begin{aligned}&\Box(A \in C_{b,a} \wedge C.CR[a,b].status = \texttt{opening} \Rightarrow \\ &\quad \Rightarrow \quad C.CR[a,b].ts \geq A.ts \geq C.CR[a,b].ts - N + 1)\end{aligned} \tag{4.16}$$

We expect the assertion (4.16) to be satisfied, because the "lifetime" of a timestamp is limited. The client keeps an incarnation for at most W_C clock ticks, the lifetime of packets in the channels is limited to L and the server also limits the lifetime of an incarnation to at most W_S clock ticks.

In case of the server, comparisons are made in *S.ClkTick*, *S.Close* and *S.RecvPkt*. The variables involved are *S.time*, *S.upper*, $S.CR[a,b].ts$ and the timestamp of the received request $R.ts$. The definition of the server assures that the following two assertions are satisfied:

$$\Box(S.CR[a,b].status \neq \texttt{closed} \Rightarrow S.time + \epsilon \geq S.CR[a,b].ts > S.time - W_S) \tag{4.17}$$

$$\Box(S.time + \epsilon \geq S.upper \geq S.time - W_S) \tag{4.18}$$

The assertions (4.17) and (4.18) and the condition below

$$W_S + \epsilon < N \tag{4.19}$$

imply the CI conditions obtained from (3.5) for the *S.ClkTick* and *S.Close* events.

The CI conditions of the *S.RecvPkt* event are given below. Each of these assertions are in the form

$$(\exists K_i : K_i < N : \Box(R \in C_{b,a} \Rightarrow P_i))$$

where P_i is a predicate on the state variables and constants. To simplify the expressions, we give only the predicates P_i ($i \in \{1, 2, 3\}$) below:

$$S.time + \epsilon + K_1 \geq R.ts \geq S.time + \epsilon + K_1 - N + 1 \tag{4.20}$$

$$S.upper + K_2 \geq R.ts \geq S.upper + K_2 - N + 1 \tag{4.21}$$

$$\begin{aligned} &S.CR[a,b].status \neq \texttt{closed} \Rightarrow \\ &\quad \Rightarrow \quad S.CR[a,b].ts + K_3 \geq R.ts \geq S.CR[a,b].ts + K_3 - N + 1 \end{aligned} \tag{4.22}$$

Without going into the details of proving the invariance of the CI conditions, it is shown here that the clocks of the client and server must be synchronized for the CI conditions to hold. This is interesting because in the specification of the protocol [LSW91] synchronization was assumed to be probabilistic. That is, most of the time the clocks should be synchronized, but there is no guarantee that they always are.

For this reason, the protocol was designed in such a way that clock synchronization affects only it s performance but it works safely even if the synchronization assumption is violated. In case of unbounded timestamp, we saw that indeed synchronization is not a precondition for the safety of the protocol. Now we are investigating the situation when modulo-N timestamps are used.

Clock synchronization for a given pair of client C and server S is captured by the following temporal formula:

$$(\exists \phi : \Box(|C.time - S.time| < \phi)) \tag{4.23}$$

Let us now assume that the system does not satisfy (4.23), i.e. there exists a computation σ such, that

$$(\forall \phi : \Diamond(|C.time - S.time| > \phi))(\sigma) \tag{4.24}$$

As a consequence of (4.24), there will be a state s_i in σ when either $C.time > S.time + \epsilon + K_1$ or $C.time < S.time + \epsilon + K_1 - N + 1$ becomes true. If at this point a *C.SendPkt*

event is executed then the CI condition (4.20) would be violated. Therefore we can conclude that the CI conditions presented above can only be satisfied if the clocks are synchronized. We note here that if the clocks are synchronized to a certain ϕ and some additional inequalities on the real-time parameters of the protocol hold, then modulo-n SCMP satisfies the safety requirements. Therefore clock synchronization is also sufficient for the safety of modulo-N SCMP, but we do not prove that statement here.

As we discussed above, it would be interesting to prove that modulo-N SCMP is safe even if the clocks are not synchronized. This is still possible since the CI conditions are *sufficient* for the safety of the modulo-N protocol, but not *necessary*. Therefore we will try to prove the safety of modulo-N SCMP in a different way. The strategy is to weaken the CI requirements (4.20)–(4.22) for the *S.RecvPkt* event in such a way that

- they are satisfied by the protocol without requiring clock synchronization,
- they still imply the required safety properties.

4.2.6 Alternative approach

We start the verification with an analysis of the behavior of the modulo-N protocol. The following case study highlights the difference between the behavior of the unbounded and modulo-N versions of the protocol. Let us assume that the clocks of the client and server are running at a constant rate, but the clock of the client is slightly faster. If we observe this system long enough, the difference between the clock of the client and the server grows beyond any limit. Now let us examine how the two protocol versions behave assuming that at the beginning $C.time = S.time$. In the following discussion, Δ will be used to denote $C.time - S.time$.

In the operation of the unbounded version, three phases can be identified (see also Figure 4.8):

Phase 1: $0 \leq \Delta < \epsilon$

During this phase the protocol operates as expected. Remember that ϵ denotes maximum *expected* clock difference and the protocol is prepared to handle differences between clocks up to this level.

Phase 2: $\epsilon \leq \Delta < \epsilon + L$

L is the maximum delay of a packet in the channels expressed in the number of clock ticks. In this phase the server *may* still accept valid requests, but it is likely to reject even previously unseen requests. It is so because the server rejects messages that have a timestamps higher than $S.time + \epsilon$. This check is necessary for proper crash recovery, for example. Whether an actual request is accepted or rejected depends on the delay of the request in the channel.

Phase 3: $\epsilon + L \leq \Delta$

No more request are accepted. Even a request that spends the maximum possible time in the channel appears to the server as too early and is thus rejected.

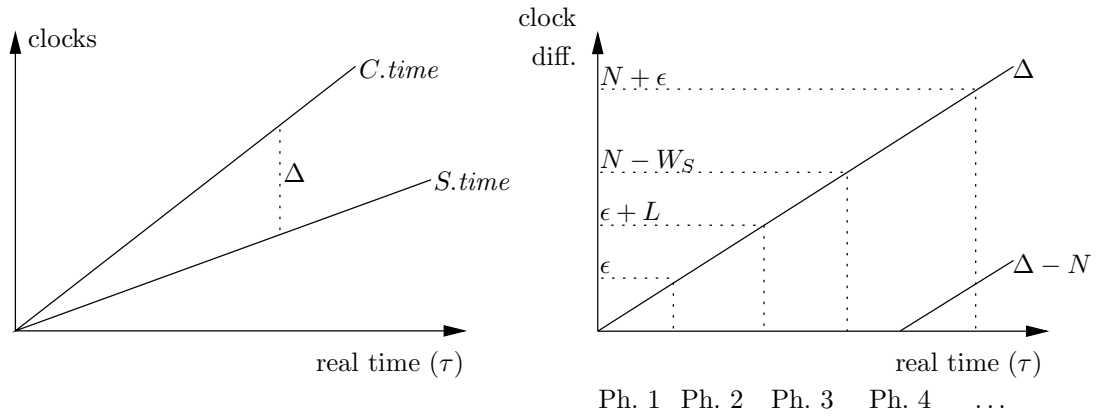


Figure 4.8: *Phases of operation at constant clock drift.*

While examining the operation of the modulo-N version of the protocol, we will assume that $\epsilon + L + W_S < N$ holds. In the first two phases the modulo-N protocol behaves identically to the unbounded protocol. Phase 3 of the unbounded protocol lasts forever, there are no more successful connection establishments after Phase 3 starts. In case of the modulo-N protocol, Phase 3 is finite because eventually Δ "wraps around" and requests become acceptable to the server again:

Phase 3a: $\epsilon + L \leq \Delta < N - W_S$

No requests are accepted similarly to the unbounded case.

Phase 4: $N - W_S \leq \Delta < N + \epsilon$

The requests from the client may be accepted again by the server. Whether an actual request is accepted depends on its delay in the channel and on the recent history of the server. If the server has a CR for the connection then a valid request is always accepted. If there is no CR, then the acceptance of the request depends on the value of *S.upper*. The value of *S.upper* may be higher than the timestamp of a valid request if the server has received higher timestamps on other connections before.

The behavior of the modulo-N protocol during Phase 4 is identical to its behavior during Phase 1. After this, the phases come in a periodic order.

To further analyze modulo-N SCMP, let us notice that the behavior of SCMP can be considered as a sliding-window protocol where the timestamps of SCMP function as sequence numbers of the sliding-window protocol. The window of the client is either of size 1 or 0. In the `opening` state, the window size is 1 because the pending request can be (re)sent. In other states no request can be sent, therefore in those states the size of the theoretical window is 0. The size of the theoretical window at the server also depends on the current state. If the state is `closed`, then the range of acceptable timestamps is $(S.upper, S.time + \epsilon]$. If the state is not `closed` then the window is $(S.CR[a, b].ts, S.time + \epsilon]$, where a and b denote the appropriate endpoint addresses.

The *similarity* between SCMP and the sliding-window protocol is that the client in

SCMP has a range of timestamps it may use and the server has a range of timestamps it is willing to accept just as the send-window in a sliding-window protocol which defines the range of sequence numbers that may be currently sent by the sender and the receive-window which defines the range of acceptable sequence numbers. The main *difference* between SCMP and the sliding-window protocol is that the relative position of the send- and receive-windows in the sliding-window protocol is maintained internally by the protocol itself (see assertion (3.6), for example), while the relative position of the windows in SCMP is determined by an external mechanism, the clock synchronization protocol.

The rules about the relative position of the windows in the sliding-window protocol assures that any old duplicate packet—either data or acknowledgment—falls outside the corresponding window and is thus rejected. These rules are maintained by advancing the window based only on packets from the peer: the receive-window is advanced as data packets are received, the send window is advanced as acknowledgments are received.

The relative position of the clocks, and of the windows, in SCMP is determined by two factors:

Clock synchronization: If we assume that the clocks are always synchronized as defined by (4.23), then the relative position of the windows is fully determined by the clock synchronization requirement.

Limited clock drift: If we assume only that the rate of the clocks is within certain bounds, then we cannot give guarantees about the relative position of the windows in general. After sufficient time, the windows can be in any position. Because of the limited drift, however, we still have guarantees about the time that is needed to change the relative position of the windows.

Based on the above observation, we can use the limited clock-drift assumption to prove the safety of modulo-N SCMP. As we saw in the case study of Figure 4.8, the lack of clock synchronization may cause that requests which would be unacceptable by unbounded SCMP are accepted by modulo-N SCMP because their timestamp "wraps into the window" of the server. Even in such cases, however, the limited drift assumption guarantees that this wrap around does not happen "too fast." If the modulo-N clocks of the client and server are synchronized at one time (Phase 1), then there is a minimum delay before the clocks may drift away so much that requests become acceptable again (Phase 4). Between these phases there is a period when no requests are accepted. If this period is so long that all the "old" packets disappear from the network during this period, then the modulo-N protocol still satisfies the safety requirements.

A weaker CI requirement which is to replace the CI requirements (4.20)–(4.22), can be informally stated as follows:

> If there is a request R in the channel from b to a, and the modulo-N timestamp of the request $R.ts$ mod N falls into the window of the server, then the unbounded timestamp $R.ts$ is higher than the timestamp of the last accepted

request. In other words,

$$\Box(R \in C_{b,a} \wedge \langle R.ts \text{ falls into the modulo-}N \text{ window}\rangle \Rightarrow \\ \Rightarrow \quad R.ts > S.CR[a,b].ts\_rcvd[S.CR[a,b].lin]) \tag{4.25}$$

In the next subsection, we extend the SCMP specification in order to be able to express this CI requirement formally.

4.2.7 Modified SCMP specification

To treat the modulo-N case formally, we will modify the specification of unbounded SCMP from the figures 4.3–4.6). The modified specification presented in this subsection models the behavior of modulo-N SCMP as it is described above. The main difference between the previous specification and this one is that the test in the server which validates a new request is carried out using modulo-N numbers.

In order to facilitate its verification, the modified specification still uses unbounded variables. The verification contains three main steps:

1. Proof that the modified specification satisfies the required safety properties (4.2)–(4.4) when using unbounded identifiers (see Section 4.2.9).
2. Specification of the CI conditions which guarantee that the specification can be implemented using modulo-N numbers (see Section 4.2.8).
3. Proof that the CI conditions are satisfied by the specification (see Section 4.2.10).

The first step is needed because this specification is different from the specification of unbounded SCMP. Because the difference is not much, we will be able to use most of the proof from Section 4.2.3 where we proved that unbounded SCMP satisfies the safety requirements. Earlier results will also be used during the second step. From the CI requirements formulated for the unbounded SCMP specification, assertions (4.14), (4.16), (4.17) and (4.18) can be used with only minor modifications. Only the CI conditions for the *S.RecvPkt* event have to be completely reformulated. The proof that the CI conditions are satisfied by the system uses real-time properties of the protocol.

The specification of the modulo-N protocol can be found in Figures 4.9–4.12. The structure of the specification is identical to the specification of the unbounded version. The modified state variables and events are explained below.

An array of epoch variables is introduced in both the client and the server processes to record the time when *ClockTick* events happen. The variable $t[i]$ records the real-time when the local time became equal to i. We assume that the limited clock drift requirement, defined by assertions (2.14) and (2.15), is satisfied by SCMP. The lower and upper bounds of the clock period are γ and Γ, respectively.

It is also assumed that the limited packet lifetime assumption (see assertion (2.13)) holds with the parameter L defining the upper bound on packet lifetimes in any of the channels.

program *ModN_SCMP*;
type
ep_addr = "type of endpoint addresses";
clnt_conn_rec = **record of**
status: {`closed`, `opening`, `open`};
ts, *lin*: **int**;
ts_sent: **array** $[0 \ldots \infty]$ **of int** $\cup$ `empty`;
open_to: **array** $[0 \ldots \infty]$ **of int** $\cup$ `empty`;
srvr_conn_rec = **record of**
status: {`closed`, `open`};
ts, *ts_srvr*, *atime*, *lin*: **int**;
ts_rcvd: **array** $[0 \ldots \infty]$ **of int** $\cup$ `empty`;
open_to: **array** $[0 \ldots \infty]$ **of int** $\cup$ `empty`;
OpenReq = **record of**
ts, *lin*: **int**;
OpenAck = **record of**
ts, *lin*, *rin*: **int**;
channel one channel $C_{a,b}$ for each a, b pair of endpoints;
Client process in Figure 4.10;
Server process in Figure 4.11;
Initial condition in Figure 4.12;

Figure 4.9: *Modulo-N SCMP: main program*

The server CR also gets two new state variables. The variable *atime* records the value of the local clock *S.time* whenever a request is accepted. The other state variable is *ts_srvr*. The value of *ts* and *ts_srvr* are both related to the timestamp from the last accepted request, but in a different way. The variable *ts* is the original timestamp from the last accepted request. The variable *ts_srvr* stores the unbounded value of the timestamp from the last request as it is interpreted by the server. *ts* and *ts_srvr* can be different when the difference between *C.time* and *S.time* exceeds N due to clock drift.

Note, however, that these two variables are always equivalent modulo-N,

$$\Box(S.CR[a,b].ts \equiv S.CR[a,b].ts\_srvr \pmod N) \tag{4.26}$$

therefore they are not distinguishable in a modulo-N implementation. The modulo-N equivalence of *ts* and *ts_srvr* is assured by the definition of *S.RecvPkt*, the only event which updates these variables. The exact role of *ts* and *ts_srvr* can be seen later when we discuss the CI requirements for the modified specification.

The modified test in the event *R.RecvPkt* for checking if a new request is acceptable looks like this:

if $R.ts \in_N (l, time + \epsilon]$ **then** ...

```
process C;
    var time: int;
        t: array[0...∞] of epoch;
        CR: array[ep_addr, ep_addr] of clnt_conn_rec;

    event ClkTick;
        when true do
            time++; t[time] := τ;
            for each local CR[a,b] do
                if CR[a,b].status ≠ closed ∧ CR[a,b].ts = time − W_C then
                    CR[a,b].status := closed;

    event Open(a,b);
        when CR[a,b].status = closed do
            CR[a,b].status := opening; CR[a,b].ts := time;
            CR[a,b].lin++; CR[a,b].ts_sent[CR[a,b].lin] := time;

    event SendPkt(a,b);
        var R: OpenReq;
        when CR[a,b].status = opening do
            R.ts := CR[a,b].ts; R.lin = CR[a,b].lin;
            send(C_{a,b}, ⟨R, τ⟩);

    event RecvPkt(a,b);
        var A: OpenAck;
            u: epoch;
        when head(C_{b,a}) = ⟨A, u⟩ do
            if CR[a,b].status = opening ∧ CR[a,b].ts = A.ts then
                CR[a,b].status := open;
                CR[a,b].open_to[CR[a,b].lin] := A.lin;

    event Close(a,b);
        when CR[a,b].status ≠ closed ∧ CR[a,b].ts < time do
            CR[a,b].status := closed;
```

Figure 4.10: *Modulo-N SCMP: client process*

```
process S;
    var time, upper: int;
        t: array[0...∞] of epoch;
        CR: array [ep_addr, ep_addr] of srvr_conn_rec;

    event ClkTick;
        when true do
            time++; t[time] := τ;
            for each local CR[a,b] do
                if CR[a,b].status ≠ closed ∧ CR[a,b].ts_srvr = time − W_S then
                    CR(a,b).status := closed;
            upper := max(upper, time − W_S);

    event RecvPkt(a,b);
        var R: OpenReq;
            u: epoch;
            l: int;
        when head(C_{b,a}) = ⟨R,u⟩ do
            if CR[a,b].status = closed then
                l := upper;
            else if CR[a,b].status ≠ closed then
                l := CR[a,b].ts_srvr;
            if R.ts ∈_N (l, time+ε] then
                CR[a,b].status := open; CR[a,b].lin++;
                CR[a,b].ts := R.ts; CR[a,b].ts_srvr := l + ((R.ts − l) mod N);
                CR[a,b].atime := time;
                CR[a,b].ts_rcvd[CR[a,b].lin] := R.ts;
                CR[a,b].open_to[CR[a,b].lin] := R.lin;

    event SendPkt(a,b);
        var A: OpenAck;
        when CR[a,b].status = open do
            A.ts := CR[a,b].ts; A.lin = CR[a,b].lin;
            A.rin = CR[a,b].open_to[CR[a,b].lin];
            send(C_{a,b}, ⟨A,τ⟩);

    event Close(a,b);
        when CR[a,b].status = open do
            upper := max(upper, CR[a,b].ts_srvr);
            CR[a,b].status := closed;
```

Figure 4.11: *Modulo-N SCMP: server process*

init

$C.time = S.time = 0 \wedge$
$(\forall a, b : a \text{ is client, } b \text{ is server :}$
$C.CR[a,b].status = S.CR[b,a].status = \texttt{closed} \wedge$
$C.CR[a,b].lin = S.CR[b,a].lin = -1 \wedge$
$C.CR[a,b].ts = S.CR[b,a].ts = S.CR[b,a].ts\_srvr = 0 \wedge$
$S.CR[b,a].atime = -1 \wedge$
$(\forall i :$
$C.CR[a,b].open\_to[i] = S.CR[b,a].open\_to[i] = \texttt{empty} \wedge$
$C.CR[a,b].ts\_sent[i] = S.CR[b,a].ts\_rcvd[i] = \texttt{empty}));$

Figure 4.12: *Modulo-N SCMP: initial condition*

This test whether $R.ts$ is in the modulo-N interval $(l, time + \epsilon]$ corresponds to the test in the sliding-window protocol that checks if the sequence number of the packet falls in the receiver window (see the *RecD* event in Figure 3.5). Formally, it is defined in the following way:

$$t \in_N (a, b] \quad = \quad t' \in [a' +_N 1, a' +_N 2, \ldots, b']$$

where $t' = t \bmod N$, $a' = a \bmod N$, and $b' = b \bmod N$. In particular, the interval is empty if $a \equiv b \pmod N$. This situation can happen in the protocol because we expect the following assertion to hold:

$$\Box(S.time + \epsilon \geq l \geq S.time - W_S)$$

4.2.8 CI requirements for the modified specification

Having specified the modified protocol, we first reformulate the CI conditions. The CI conditions (4.14), (4.16), and (4.18) can be used without modifications. Only the CI condition (4.17) has to be slightly modified by replacing $S.CR(a,b).ts$ with the new state variable $S.CR(a,b).ts\_srvr$. The CI condition then becomes (4.29). For a complete reference, we list below all the CI conditions:

$$\Box(C.CR[a,b].status \neq \texttt{closed} \Rightarrow$$
$$\Rightarrow \quad C.CR[a,b].time \geq C.CR[a,b].ts > C.CR[a,b].time - W_C) \tag{4.27}$$

$$\Box(A \in C_{b,a} \wedge C.CR[a,b].status = \texttt{opening} \Rightarrow$$
$$\Rightarrow \quad C.CR[a,b].ts \geq A.ts \geq C.CR[a,b].ts - N + 1) \tag{4.28}$$

$$\Box(S.CR[a,b].status \neq \texttt{closed} \Rightarrow$$
$$\Rightarrow \quad S.time + \epsilon \geq S.CR[a,b].ts\_srvr > S.time - W_S) \tag{4.29}$$

$$\Box(S.time + \epsilon \geq S.upper \geq S.time - W_S) \tag{4.30}$$

The modified CI condition (4.29) illustrates the role of *ts* and *ts_srvr* in the server's CR. When formulating CI conditions for the modified specification, *ts* and *ts_srvr* can be used alternately depending on the other variables involved. When the timestamp stored by the server is compared to variables that have their value from the server's clock, then we use *ts_srvr* in the CI condition. This is the case for the comparisons in the *S.ClkTick* and *S.Close* events. On the other hand, when sending an acknowledgment A, then $A.ts$ is filled in from *ts* in order to assure that the timestamp carried by A is originating from the client's clock.

The CI conditions listed above cover the modulo-N tests in all events but *S.RecvPkt*. The CI condition (4.25) for this event is informally stated at the end of the previous subsection. Using the modified definition, the CI condition becomes

$$\Box((R \in C_{b,a} \wedge \{R.ts \in (l, S.time + \epsilon]\} \bmod N) \Rightarrow$$
$$\Rightarrow \quad R.ts > S.CR[a,b].ts\_rcvd[S.CR[a,b].lin])$$

where l is the local variable of the *R.RecvPkt* event. Using the specification of the event, the above assertion can be replaced by the following two assertions when substituting the value of l determined by the current state of the server:

$$\Box((R \in C_{b,a} \wedge S.CR[a,b].status \neq \texttt{closed} \wedge$$
$$\wedge\{R.ts \in (S.CR[a,b].ts\_srvr, S.time + \epsilon]\} \bmod N) \Rightarrow \tag{4.31}$$
$$\Rightarrow \quad R.ts > S.CR[a,b].ts\_rcvd[S.CR[a,b].lin])$$

$$\Box((R \in C_{b,a} \wedge S.CR[a,b].status = \texttt{closed} \wedge$$
$$\wedge\{R.ts \in (S.CR[a,b].upper, S.time + \epsilon]\} \bmod N) \Rightarrow \tag{4.32}$$
$$\Rightarrow \quad R.ts > S.CR[a,b].ts\_rcvd[S.CR[a,b].lin])$$

4.2.9 Safety using unbounded timestamps

Lemma 4.3 *The modified SCMP specification satisfies the safety requirements defined by the assertions (4.2)–(4.4) provided that the assertions (4.31) and (4.32) are satisfied by the system.*

The proof of Lemma 4.3 is relatively straightforward, because we can use most of the proof that the original SCMP specification satisfies these requirements. The only substantial difference is in the way of proving that $S.CR[a,b].ts\_rcvd$ is monotone increasing which is stated by assertion (4.9). For unbounded SCMP, we formulated assertion (4.12) and (4.13) as a precondition of (4.9) with respect to the *S.RecvPkt* event. In case of the modified specification, the CI conditions (4.31) and (4.32) are the preconditions of assertion (4.9) with respect to the *S.RecvPkt* event. Therefore, (4.9) is preserved by the event provided that (4.31) and (4.32) are satisfied by the system. The assertions (4.12) and (4.13) are not necessary, furthermore assertion (4.13) does not even hold for the modified SCMP specification.

4.2.10 Invariance of the CI requirements

Lemma 4.4 *The modified specification satisfies its CI requirements defined by assertions (4.27)–(4.32).*

The invariance of (4.27), (4.29) and (4.30) can be proven from the specification of modulo-N SCMP without using the real-time properties for the system. These proofs are relatively straightforward.

On the other hand, the invariance of (4.28), (4.31) and (4.32) can only be proven by exploiting the real-time properties of the protocol. In particular, we will use the maximum packet-lifetime assumption (2.13) and the limited clock-drift assumption (2.14) and (2.15). An informal argument based on operational reasoning is given here to give an idea why these assertions are satisfied by the system.

The CI condition (4.28) states that if there is an acknowledgment packet A in the channel, then its associated timestamp cannot be arbitrary. It is easy to prove that

$$\Box(A \in C_{b,a} \Rightarrow C.CR[a,b].ts \geq A.ts)$$

This assertion holds simply because $C.time$ is monotone increasing. As for the lower limit on $A.ts$, we will prove a stronger assertion

$$\Box(A \in C_{b,a} \Rightarrow A.ts \geq C.time - N + 1)$$

This assertion is expected to hold because the client and the server both limit the lifetime of an incarnation and the lifetime of packets in the channels is also limited.

Figure 4.13 illustrates this in an example. An incarnation is started at time t_0 when $C.time = n$. The client sends a request R at time t_1, the request is received and accepted at t_2 by the server, at t_3 the server sends back an acknowledgment which is received at t_4 by the client. Note that although all the intervals are depicted on the right side of the

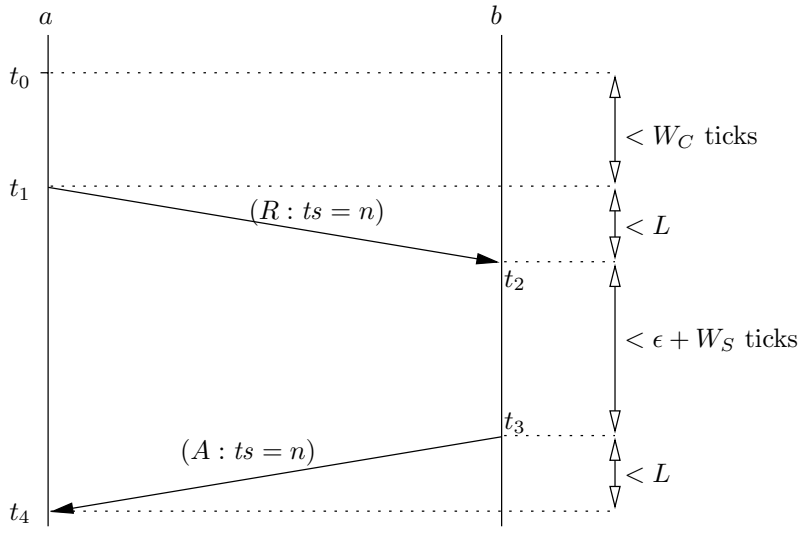


Figure 4.13: *Oldest possible acknowledgment*

figure, the W_C and W_S clock ticks are measured on the clock of the client and server, respectively.

At t_1 the client's clock is less than $R.ts + W_C$ because the client closes an incarnation if it becomes W_C ticks old. The maximum packet-lifetime assumption (2.13) implies that $t_2 - t_1 < L$ and $t_4 - t_3 < L$. The server keeps an incarnation open for at most $\epsilon + W_S$ ticks. From the upper limit on the time between two ticks of the server's clock (2.15), we know that $t_3 - t_2 < (\epsilon + W_S) \cdot \Gamma$. Therefore $t_4 - t_1 < 2L + (\epsilon + W_S) \cdot \Gamma$. The clock of the client advances at most $(t_4 - t_1)/\gamma$ ticks between t_1 and t_4. Combining all these together, we get that

$$\Box(A \in C_{b,a} \Rightarrow A.ts + W_C + \frac{2L + (\epsilon + W_S) \cdot \Gamma}{\gamma} > C.time)$$

From the above assertion we can see that (4.33) is a sufficient condition for the invariance of (4.28)

$$N > W_C + \frac{2L + (\epsilon + W_S) \cdot \Gamma}{\gamma} \tag{4.33}$$

The CI conditions (4.31) and (4.32) both state that if the timestamp of a request is in the modulo-N window of the server, then the unbounded timestamp of the request must be greater than the timestamp of the last accepted request. To show how these CI conditions can be enforced, we will construct situations when the conditions would be violated and then find sufficient conditions that exclude these situations.

Specifically, we consider the case shown in Figure 4.14. The server accepts a request R at time t_3. The question is "which is the oldest possible timestamp still in the channel

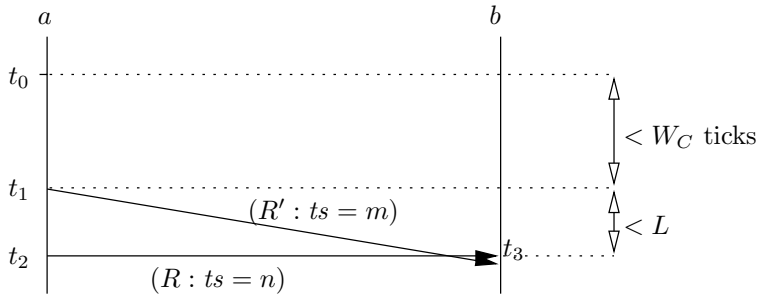


Figure 4.14: *Oldest possible request*

at the moment of t_3?" The request accepted at t_3 was sent at t_2, therefore its timestamp $R.ts$ is less than or equal to the value of the client's clock $C.time$ at t_2. It is evident that $t_2 \leq t_3$.

Let us assume that the request R' carrying the oldest timestamp in the channel was transmitted at t_1. Because R' is still in the channel, $t_3 - t_1 < L$ must hold. $R'.ts$ carries the value of the client's clock from t_0 when this incarnation was started. The client does not keep an incarnation open for more than W_C clock ticks. Combining all these, we get that

$$R.ts - R'.ts < W_C + \frac{L}{\gamma}$$

Because R is accepted at t_3, $R.ts$ must be within the server's window. This window is not larger than $[S.time - W_S, S.time + \epsilon]$. To avoid the acceptance of R', its timestamp must be outside of the window. This can be achieved if

$$R.ts - R'.ts < N - (\epsilon + W_S)$$

A pictorial explanation of this inequality can be found in Figure 4.15. The requirement is that even if the newest timestamp $R.ts$ appears to be in the lowest part of the window, the oldest timestamp $R'.ts$ must not wrap into the window from the other end. This is assured by the above inequality.

Therefore, combining the above two inequalities, we get that (4.34) below is a sufficient condition for the invariance of the CI conditions (4.31) and (4.32).

$$N > W_C + \frac{L}{\gamma} + \epsilon + W_S \tag{4.34}$$

This completes the argument that the CI conditions of the modified SCMP specification are satisfied by the system. Of course, the above argument was not intended to replace a rigorous proof, the proof of the invariance of the CI conditions based on temporal logic can be found in Appendix B.

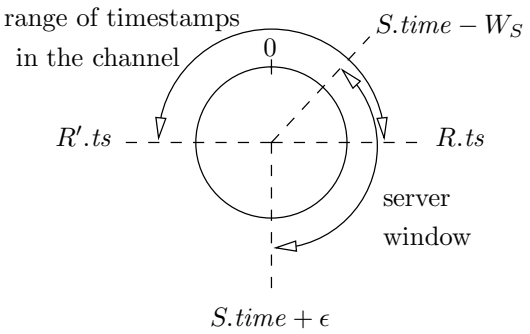


Figure 4.15: *The server window and the range of timestamps in the channel*

Theorem 4.5 *The modified SCMP specification satisfies the desired safety requirements (4.2)–(4.4) even when using modulo-N identifiers provided that constraints (4.33) and (4.34) are met.*

Theorem 4.5, which is our main result, follows from Lemma 4.3 and 4.4.

Let us summarize the results of this section. First we have shown, that if we want to implement the unbounded SCMP specification from Section 4.2.1 using modulo-N identifiers, then we must assume synchronized clocks. Then we have constructed a modified specification (see Section 4.2.7) and we have proven that this specification of SCMP

- satisfies the safety requirements;
- can be implemented using modulo-N identifiers without assuming clock synchronization.

The important result of this verification is that the safety of SCMP does not necessarily depend on the synchronization of the clocks. On the other hand, the dependency on the existence of maximum packet-lifetimes and on the limited clock drift is not unique to SCMP. In fact, we expect that every protocol which uses modulo-N identifiers over a channel which may reorder packets depends on these assumptions.

4.2.11 Modeling of failures

A CM protocol must be prepared for the failure of hosts which is usually called a crash. When a host crashes, it loses the contents of its memory and stops working. The actions taken by a host to recover from crashes should preserve the safety requirements and should assure that normal communication commences as soon as possible.

To model a crash, we have to introduce a new state variable *rstatus* which can be either `running` or `down`; and two new events *Crash* and *Recover*. Then we add *rstatus* = `running` to the enabling condition of the events in Figure 4.4 and 4.5 except the *ClkTick* events. We assume that the clocks do not stop working, or if they do then they can be reinitialized after the crash.

event *Crash* { The client and server events are identical. }
    when *rstatus* = `running` **do**
        rstatus := `down`;

event *C.Recover*
    when *rstatus* = `down` and "*time* has advanced since the crash" **do**
        rstatus := `running`;
        for each $CR[a, b]$ **do**
            $CR[a, b].status$:= `closed`;

event *R.Recover*
    when *rstatus* = `down` **do**
        rstatus := `running`;
        upper := $time + \epsilon$;
        for each $CR[a, b]$ **do**
            $CR[a, b].status$:= `closed`;

Figure 4.16: *Crash recovery at the client and at the server*

The specification of the *Crash* and *Recover* events is in Figure 4.16. *Crash* simply sets *rstatus* to `down` for both clients and servers. A client is only allowed to recover from a crash when its clock advanced at least by one tick. This restriction is necessary because the status of every connection is set to `closed` during the recovery and assertion (4.10) has to be preserved.

A server is allowed to recover immediately after a crash. Setting *upper* to $time + \epsilon$, the monotonicity of *time* and the assertion below

$$\Box(S.CR[a, b].ts\_rcvd[S.CR[a, b].lin] \leq time + \epsilon)$$

assure that (4.12) will be preserved by the recovery procedure thus no duplicate requests will be accepted.

Based on the above hints, it is trivial to extend the verifications for the extended protocol which models crashes. The constraints for the safety of the modulo-N protocol remain valid. There is another crash-recovery mechanism discussed by [LSW91] which requires safe storage for a server state variable. That mechanism can be modeled in a similar way.

4.3 Hybrid 2WHS and 3WHS protocols

The protocols discussed in this section use either 2WHS or 3WHS to open a connection depending on the current protocol state. In this respect, they are different from SCMP

which always opens connections with 2WHS. The advantage of these hybrid protocols is that they can use the available data storage as a cache: if there is information about the connecting peer in the cache then the request can be validated without the 3WHS, otherwise a 3WHS must be performed. This allows to trade memory for connection setup latency. Connection information can be discarded from the cache to free memory, but this will increase the probability that subsequent connection setups must go through the 3WHS.

The discussion is based on the family of protocols presented by Shankar and Lee in [SL93, SL95]. The protocols are called SC (server cache). Modifications to the specification and refinements to the analysis were proposed by the author in [OHdG96b]. These modifications and some others will be discussed here.

The caching in these protocols is such that there is always a minimum required caching period. Data must remain in the cache for at least this period, otherwise the protocol can fail. After this minimum period, data can be discarded freely. One of our proposals discussed here is a technique to shorten the minimum caching period in most situations. The technique is called implicit 3WHS and it will be described in Section 4.3.4.

The correctness conditions for SC include the maximum incarnation lifetime I. That is, the correctness of the protocol is not guaranteed without a bound on the length of connections. This is an undesirable property because connections are opened and closed by users of the transport service and it usually it is difficult or even impossible to enforce a maximum connection duration. Another caching technique is proposed in [SL95] to eliminate the dependency on I. We propose an alternative solution which combines the connection management and data transfer protocols. The advantage of our technique is that it does not require additional memory.

4.3.1 Basic protocol

The first part of the specification *SC* can be found in Figure 4.17. No formal verification of *SC* is presented in the thesis. Because of this and in order to simplify the discussion, auxiliary variables are not shown in this specification. Extending the specification with the necessary auxiliary variables is straightforward if one wants to carry out a correctness proof like the one presented in Section 4.2.

State variables and message formats

Similarly to SCMP, the protocol specification consists of clients C and servers S. Both clients and servers maintain their status information in connection records CR. A client maintains the following state variables in its connection record:

- *Status*: The status of the connection from the client's standpoint. `closed` if the client has no active connection; `opening` if the client has a pending open request;

program *SC*;
type
ep_addr = ⟨type of endpoint addresses⟩;
clnt_conn_rec = **record of**
Status: {`closed`, `opening`, `opnclng`, `open`, `closing`};
Lin, *Din*: {`nil`} ∪ {0, 1, ...};
srvr_conn_rec = **record of**
Status: {`closed`, `opening`, `open`, `closing`};
Lin, *Din*: {`nil`} ∪ {0, 1, ...};
Cache: {`old`, `nil`} ∪ {0, 1, ...};
CRQ, *DRQ* = **record of**
sin: {0, 1, ...};
CRR, *CRA*, *CRRA*, *DRA* = **record of**
sin, *rin*: {0, 1, ...};
REJ = **record of**
rin: {0, 1, ...};

channel ⟨one channel $C_{a,b}$ for each a, b pair of endpoints⟩;

process *C*;
var *CR*: **array** [*ep_addr*, *ep_addr*] **of** *clnt_conn_rec*;
LinGen: **array** [*ep_addr*] **of** {0, 1, ...};
Client events are specified in Figure 4.18 and 4.19;

process *S*;
var *CR*: **array** [*ep_addr*, *ep_addr*] **of** *srvr_conn_rec*;
LinGen: **array** [*ep_addr*] **of** {0, 1, ...};
Server events are specified in Figure 4.20, 4.21, and 4.22;

Figure 4.17: *SC: main program*

`opnclng` if the client's open request is not acked yet, but the user has already indicated its intention to close the connection; `open` if the connection is established; and `closing` if the client has a pending close request.

- *Lin*: The local incarnation number. *Lin* is `nil` if the client is `closed`, otherwise it identifies the incarnation of the client side involved in the current connection.
- *Din*: The distant incarnation number. Its value is `nil` if the client's open request has not been acked yet, otherwise it identifies the incarnation of the server side involved in the connection.

Furthermore, there is an incarnation number generator *LinGen*. Unique incarnation numbers are taken from this generator. In implementations of *SC*, incarnation numbers can be generated either by a counter as it is done in the specification here, or by a

monotone clock. The only requirement is that *LinGen* is not incremented faster than a certain rate in order to maintain protocol correctness (see (4.48) below). All connections originating from a client endpoint a share the generator $LinGen[a]$.

The server maintains similar variables in its connection record with the following differences:

- *Status* has different values. The `opening` state means that the server is currently involved in a 3WHS with the client. The `opnclng` state has no meaning for the server. The server enters the `closing` state when the client requests a disconnect.
- *Cache*: Normally, it stores the last incarnation number of the client. The incarnation number in open requests is compared to this. *Cache* is `nil` if nothing is known about the last connection attempt from the client and a 3WHS is needed to open a new connection. The value `old` means that the last value stored in the cache has expired but no connection has been accepted from the client since then. Any open request can be accepted with the 2WHS in this case.

Messages are identified by their type and by the numbers of the client and server incarnations. The following message types are sent by clients:

- *CRQ*: Connection request, sent when opening.
- *CRRA*: Acknowledgment to a *CRR* message to complete the 3WHS.
- *DRQ*: Disconnect request.
- *REJ*: Reject, sent in response to an unacceptable message.

The list of message types send by servers:

- *CRR*: Reply to a *CRQ* when 3WHS is used.
- *CRA*: Acknowledgment to a *CRQ* when 2WHS is used.
- *DRA*: Acknowledgment to a *DRQ* when closing.
- *REJ*: Reject, sent in response to an unacceptable message.

In all message types the header field *sin* carries the local incarnation number of the sender, *rin* carries the remote incarnation number of the intended receiver. Some of the fields are missing in certain message types.

Messages can also be classified as primary or secondary. Primary messages are sent repeatedly until a response is received or a timeout occurs. Secondary messages can only be sent in response to the reception of a primary message. It is also possible that a primary message is sent in response to a primary message. This is the case in the 3WHS when a *CRR* is sent in response to a *CRQ*.

Client events

Figure 4.18 lists the client events related to user actions and the events to send primary messages. Figure 4.19 list the receive events of the client.

event $ConnReq(a, b)$;
  when $CR[a, b].Status = \texttt{closed}$ **then**
    $CR[a, b].Status := \texttt{opening}$; $CR[a, b].Lin := LinGen[a] + +$;
event $DiscReq(a, b)$;
  when $CR[a, b].Status \in \{\texttt{opening}, \texttt{open}\}$ **then**
    if $CR[a, b].Status = \texttt{opening}$ **then** $CR[a, b].Status := \texttt{opnclng}$;
    else $CR[a, b].Status := \texttt{closing}$;
event $Abort(a, b)$;
  when $CR[a, b].Status \neq \texttt{closed} \wedge \langle$no resp. for more than W_C sec$\rangle$ **then**
    $CR[a, b].Status := \texttt{closed}$; $CR[a, b].Lin := \texttt{nil}$; $CR[a, b].Din := \texttt{nil}$;
event $SendCR(a, b)$;
  when $CR[a, b].Status \in \{\texttt{opening}, \texttt{opnclng}\}$ **then**
    $CRQ.sin := CR[a, b].Lin$; **send**$(C_{a,b}, CRQ)$;
event $SendDR(a, b)$;
  when $CR[a, b].Status \in \{\texttt{opnclng}, \texttt{closing}\}$ **then**
    $DRQ.sin := CR[a, b].Lin$; **send**$(C_{a,b}, DRQ)$;

Figure 4.18: *SC: client events, part 1*

The event *ConnReq* is triggered by the user when it wants to open a new connection. The status is changed to `opening` and an incarnation number is generated. Another user-activated event is *DiscReq*. This event indicates that the user wants to close the connection. Depending on the current state of the client, the state becomes either `closing` or `opnclng`. The *Abort* event is triggered by a timeout. If a response is outstanding for more than W_C seconds then the connection is forcibly closed. The bound W_C on the retransmission period of *CRQ* is required by protocol correctness.

The states `opening` and `opnclng` indicate that an ack of the client's *CRQ* message is expected. Sending a *CRQ* message is thus enabled in these states (event *SendCR*). Similarly, *DRQ* messages can be generated in the states `opnclng` and `closing`.

CRR messages are received in response to a *CRQ* if the server has no cached information to validate the request. The first clause in *RecvCRR* event processes *CRR* messages that acknowledge a previously sent *CRQ*. The state is changed to `open` or `closing`, the server incarnation is recorded in *Din* and a *CRRA* message is sent to complete the 3WHS. The second clause processes duplicate *CRR* messages. A *CRRA* message is sent in response. The third clause can happen after a server crash or early connection discard. $CRR.sin > CR[a, b].Din$ indicates that we are connected to an old server incarnation. The connection is closed and a *REJ* is sent. The same is done if a *CRR* is received in the `closed` state.

A *CRA* message with the proper incarnation number in `opening` and `opnclng` states indicates a successful 2WHS. The state is changed and the remote incarnation identifier is stored *Din*. *DRA* messages are accepted in `closing` state only if they have the proper

event $RecvCRR(a, b)$;
  when head$(C_{b,a}) = CRR$ **then** $CRR :=$ **recv**$(C_{b,a})$;
    if $CR[a, b].Status \in \{\texttt{opening}, \texttt{opnclng}\} \wedge CRR.rin = CR[a, b].Lin$ **then**
      if $CR[a, b].Status = \texttt{opening}$ **then** $CR[a, b].Status := \texttt{open}$;
      else $CR[a, b].Status := \texttt{closing}$;
      $CR[a, b].Din := CRR.sin$;
      $CRRA.sin := CR[a, b].Lin$; $CRRA.rin := CR[a, b].Din$; **send**$(C_{a,b}, CRRA)$;
    elif $CR[a, b].Status \in \{\texttt{open}, \texttt{closing}\} \wedge CRR.rin = CR[a, b].Lin \wedge$
        $CRR.sin = CR[a, b].Din$ **then**
      $CRRA.sin := CR[a, b].Lin$; $CRRA.rin := CR[a, b].Din$; **send**$(C_{a,b}, CRRA)$;
    elif $CR[a, b].Status \in \{\texttt{open}, \texttt{closing}\} \wedge CRR.rin = CR[a, b].Lin \wedge$
        $CRR.sin > CR[a, b].Din$ **then**
      $CR[a, b].Status := \texttt{closed}$; $CR[a, b].Lin := \texttt{nil}$; $CR[a, b].Din := \texttt{nil}$;
      $REJ.rin := CRR.sin$; **send**$(C_{a,b}, REJ)$;
    elif $CR[a, b].Status = \texttt{closed}$ **then**
      $REJ.rin := CRR.sin$; **send**$(C_{a,b}, REJ)$;
event $RecvCRA(a, b)$;
  when head$(C_{b,a}) = CRA$ **then** $CRA :=$ **recv**$(C_{b,a})$;
    if $CR[a, b]Status \in \{\texttt{opening}, \texttt{opnclng}\} \wedge CRA.rin = CR[a, b].Lin$ **then**
      if $CR[a, b].Status = \texttt{opening}$ **then** $CR[a, b].Status := \texttt{open}$;
      else $CR[a, b].Status := \texttt{closing}$;
      $CR[a, b].Din := CRA.sin$;
    elif $CR[a, b].Status \in \{\texttt{open}, \texttt{closing}, \texttt{closed}\}$ **then** ⟨no action⟩;
event $RecvDRA(a, b)$;
  when head$(C_{b,a}) = DRA$ **then** $DRA :=$ **recv**$(C_{b,a})$;
    if $CR[a, b]Status = \texttt{closing} \wedge DRA.rin = CR[a, b].Lin \wedge$
        $DRA.sin = CR[a, b].Din$ **then**
      $CR[a, b].Status := \texttt{closed}$; $CR[a, b].Lin := \texttt{nil}$; $CR[a, b].Din := \texttt{nil}$;
    elif $CR[a, b].Status \in \{\texttt{opening}, \texttt{opnclng}, \texttt{open}, \texttt{closed}\}$ **then** ⟨no action⟩;
event $RecvREJ(a, b)$;
  when head$(C_{b,a}) = REJ$ **then** $REJ :=$ **recv**$(C_{b,a})$;
    if $CR[a, b]Status \in \{\texttt{opening}, \texttt{opnclng}, \texttt{closing}\} \wedge REJ.rin = CR[a, b].Lin$ **then**
      $CR[a, b].Status := \texttt{closed}$; $CR[a, b].Lin := \texttt{nil}$; $CR[a, b].Din := \texttt{nil}$;
    elif $CR[a, b].Status \in \{\texttt{open}, \texttt{closed}\}$ **then** ⟨no action⟩;

Figure 4.19: *SC: client events, part 2*

event *MakeNil*(b, a);
  when $CR[b, a].Cache \notin \{\texttt{nil}, \texttt{old}\} \wedge$
    ⟨entry was updated between $c_S^{(\texttt{nil})}$ and $c_S^{(\texttt{old})}$ seconds ago⟩ **then**
    $CR[b, a].Cache := \texttt{nil}$;
event *MakeOld*(b, a);
  when $CR[b, a].Cache \notin \{\texttt{nil}, \texttt{old}\} \wedge$
    ⟨entry was updated between $c_S^{(\texttt{old})}$ and C_S seconds ago⟩ **then**
    $CR[b, a].Cache := \texttt{old}$;
event *Release*(b, a);
  when $CR[b, a].Status = \texttt{closing} \wedge$
    ⟨client had sufficient time to ask for retransmissions⟩ **then**
    $CR[b, a].Status := \texttt{closed}$; $CR[b, a].Lin := \texttt{nil}$; $CR[b, a].Din := \texttt{nil}$;
event *Abort*(b, a);
  when $CR[b, a].Status \neq \texttt{closed} \wedge$ ⟨no resp. for more than W_S sec⟩ **then**
    $CR[b, a].Status := \texttt{closed}$; $CR[b, a].Lin := \texttt{nil}$; $CR[b, a].Din := \texttt{nil}$;
event *SendCRR*(b, a);
  when $CR[b, a].Status = \texttt{opening}$ **then**
    $CRR.sin := CR[b, a].Lin$; $CRR.rin := CR[b, a].Din$; **send**$(C_{b,a}, CRR)$;

Figure 4.20: *SC: server events, part 1*

incarnation identifiers. Although *DRQ* messages can also be generated in the `opnclng` state, the corresponding *DRA* can only be accepted if the preceding *CRA* has already been received and the state has moved to `closing`. If the 3WHS is used, then the *DRA* can never overtake the *CRR* message. A *REJ* message causes the current connection to be aborted.

Server events

Figure 4.20 contains the non-receive events of server S, the receive events are listed in Figure 4.21 and 4.22.

The *MakeNil* and *MakeOld* events specify the constraints on the caching policy. Each entry must stay in the cache for at least $c_S^{(\texttt{nil})}$ seconds. When this minimum caching period has expired, the entry may be set to `nil`. If the entry was longer than $c_S^{(\texttt{old})}$ in the cache, then it may be set to `old`. This is preferred because connections are opened with a 2WHS when $CR[b, a].Cache = \texttt{old}$. If modulo-$N$ incarnation numbers are used, then it is also important that an entry is not kept in the cache longer than the maximum caching period C_S.

The event *Release* closes a connection in the `closing` state. The length of staying in the `closing` state must be long enough that the client can retransmit the *DRQ* message if the *DRA* is lost. The server side of the connection is closed by a timer because the last

```
event RecvCR(b, a);
    when head(C_{a,b}) = CRQ then CRQ := recv(C_{a,b});
        if CR[b, a].Status = closed ∧ ⟨rejecting connections⟩ then
            REJ.rin := CRQ.sin; send(C_{b,a}, REJ);
            if CR[b, a].Cache = old ∨ CRQ.sin > CR[b, a].Cache ≠ nil then
                CR[b, a].Cache := CRQ.sin;
        elif CR[b, a].Status = closed ∧ ⟨accepting connections⟩ ∧
                CR[b, a].Cache = nil then
            CR[b, a].Status := opening; CR[b, a].Lin := LinGen[b] + +;
            CR[b, a].Din := CRQ.sin;
        elif CR[b, a].Status ∈ {closing, closed} ∧ ⟨accepting connections⟩ ∧
                (CR[b, a].Cache = old ∨ CRQ.sin > CR[b, a].Cache ≠ nil) then
            CR[b, a].Status := open; CR[b, a].Lin := LinGen[b] + +;
            CR[b, a].Din := CRQ.sin; CR[b, a].Cache := CRQ.sin;
            CRA.sin := CR[b, a].Lin; CRA.rin := CR[b, a].Din; send(C_{b,a}, CRA);
        elif CR[b, a].Status = opening ∧ CRQ.sin > CR[b, a].Din then
            CR[b, a].Lin := LinGen[b] + +; CR[b, a].Din := CRQ.sin;
        elif CR[b, a].Status = open ∧
                (CR[b, a].Cache = old ∨ CRQ.sin > CR[b, a].Cache ≠ nil) then
            if ⟨willing to reopen⟩ then
                CR[b, a].Lin := LinGen[b] + +; CR[b, a].Din := CRQ.sin;
                CR[b, a].Cache := CRQ.sin;
                CRA.sin := CR[b, a].Lin; CRA.rin := CR[b, a].Din; send(C_{b,a}, CRA);
            else
                CR[b, a].Status := closed; CR[b, a].Lin := nil; CR[b, a].Din := nil;
                CR[b, a].Cache := CRQ.sin;
        elif CR[b, a].Status = open ∧ CRQ.sin = CR[b, a].Cache ∉ {nil, old} then
            CRA.sin := CR[b, a].Lin; CRA.rin := CR[b, a].Din; send(C_{b,a}, CRA);
```

Figure 4.21: *SC: server events, part 2*

message (*DRA*) in a connection is sent by the server.

The server's *Abort* event is equivalent to the client's same event. In the `opening` state the server generates *CRR* messages (event *SendCRR*) until a *CRRA* is received which moves the state to `open` or the connection is timed out.

The *RecvCR* event of the server is shown in Figure 4.21. The first clause specifies the processing of *CRQ* messages when the server is not accepting new connection requests. Note that it is the user of the transport service who decides whether the server is accepting or rejecting connections. A *REJ* is sent in response to the *CRQ*, but *Cache* is updated if the request would otherwise be acceptable.

The second clause is taken when there is no cached entry and a 3WHS is needed. The third clause happens when the connection can be opened by a 2WHS. If the server

```
event RecvCRRA(b, a);
    when head(C_{a,b}) = CRRA then CRRA := recv(C_{a,b});
        if CR[b, a].Status = opening ∧ CRRA.sin = CR[b, a].Din ∧
                CRRA.rin = CR[b, a].Lin then
            CR[b, a].Status := open; CR[b, a].Cache := CRRA.sin;
        elif CR[b, a].Status ∈ {open, closing, closed} then ⟨no action⟩;
event RecvDR(b, a);
    when head(C_{a,b}) = DRQ then DRQ := recv(C_{a,b});
        if CR[b, a].Status = {open, closing} ∧ DRQ.sin = CR[b, a].Din then
            if CR[b, a].Status = open then CR[b, a].Status := closing;
            DRA.sin := CR[b, a].Lin; DRA.rin := CR[b, a].Din; send(C_{b,a}, DRA);
        elif CR[b, a].Status = closed then
            REJ.rin := DRQ.sin; send(C_{b,a}, REJ);
        elif CR[b, a].Status = opening then ⟨no action⟩;
event RecvREJ(b, a);
    when head(C_{a,b}) = REJ then REJ := recv(C_{a,b});
        if CR[b, a].Status = opening ∧ REJ.rin = CR[b, a].Lin then
            CR[b, a].Status := closed; CR[b, a].Lin := nil;
        elif CR[b, a].Status ∈ {open, closing, closed} then ⟨no action⟩;
```

Figure 4.22: *SC: server events, part 3*

is still in the `closing` state, then the client reopens the connection before the server released the previous connection record. Since $CRQ.sin > CR[b, a].Cache$ indicates that the message is new, it is considered as an implicit acknowledgment of the server's *DRA* message in the previous connection.

The fourth clause happens if the server starts a 3WHS in response to an old *CRQ* message and then receives a recent *CRQ*. Since the connection has not moved beyond the `opening` state, the server simply reconnects to the new client incarnation and restarts the 3WHS. Generating a new server incarnation is important because the client may have become connected to the old incarnation already.

The fifth clause shows the situation when the client reopens after a timeout or crash. The server opens a new connection if the user is willing to reconnect otherwise it simply discards the old connection. Finally, the sixth clause describes the response to a duplicate *CRQ*.

The reception of a *CRRA* message in the `opening` state causes the server to change its state to `open`. A *DRQ* is replied by a *DRA* is the `open` and `closing` states. If necessary, the state is also changed to `closing`. *DRQ* messages are replied by a *REJ* in the closed state. In the `opening` state no action is taken. *DRQ* messages in this state can be caused by the client receiving a *DiscReq* event before the connection becomes established.

The typical opening sequences are shown in Figure 4.23 and 4.24. The first figure shows

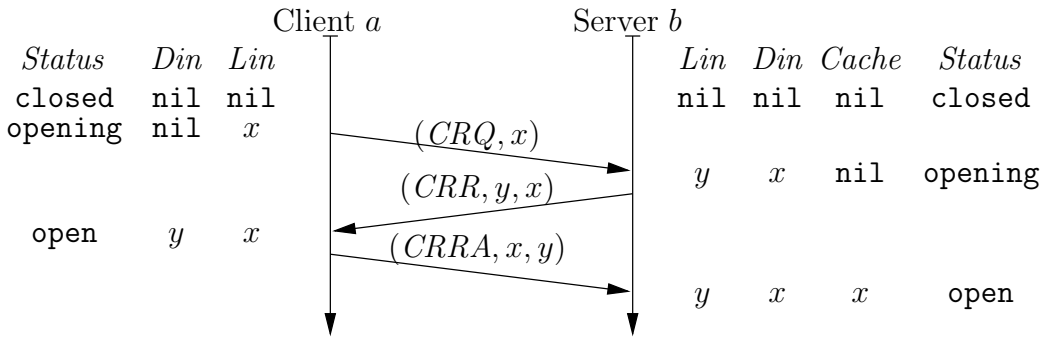


Figure 4.23: *Opening a connection by 3WHS*

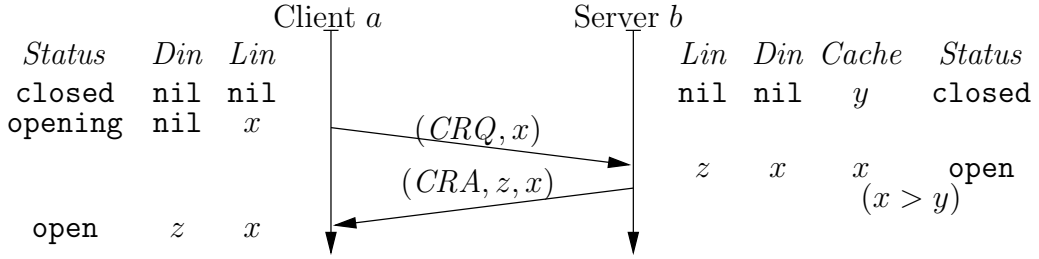


Figure 4.24: *Opening a connection by 2WHS*

a 3WHS between client a and server b. The notation (M, sin, rin) stands for a message of type M with the appropriate header fields. The other figure shows the situation when the server has an entry in its cache for the client. In this case a 2WHS can be used to open the connection.

4.3.2 Protocol properties

Although the auxiliary variables are not included in the *SC* specification which were used to formally specify the desired property (4.1) in Section 4.1, the desired property can stated informally. If incarnation x of endpoint c becomes open to incarnation y of endpoint d, i.e., $CR[c,d].Status = \texttt{open}$, $CR[c,d].Lin = x$, and $CR[c,d].Din = y$, then incarnation y of d is either open to x of c, or y of d never becomes open to any incarnation.

These correctness constraints are satisfied by *SC* if the following condition is satisfied [OHdG96b, SL95]:

$$c_S^{(\texttt{nil})} > W_C \quad \wedge \quad c_S^{(\texttt{old})} > W_C + L \tag{4.35}$$

In words, this means that the minimum caching period $c_S^{(\texttt{nil})}$ of the server must be longer than the retransmission timeout W_C of the client. The cache entry can be replaced by `old` if it stayed in the cache longer than the client's retransmission timeout plus the maximum

packet lifetime L. Some other constraints are determined in [SL95] and [OHdG96b], but those are only needed if we want to model crashes (see also Section 4.2.11).

In order to implement the protocol using modulo-N incarnation numbers, correct interpretation conditions must be formulated. It can be done using (3.5) repeatedly for each event where modulo-N identifiers are compared. Without formulating these CI conditions, we list assertions that can be used to deduce the bound K of (3.5) in these CI conditions. The real-time conditions for the correctness of the protocol are obtained from these bounds.

Modulo-N incarnation numbers cannot be used without an upper bound on the rate of generating new incarnation number. The parameter α denotes the minimum time between two incarnation generations. That is, *LinGen* must not be read and incremented faster than the rate $1/\alpha$. The constraint may be explicitly enforced by the implementation or α must be chosen such that it is physically impossible to exceed this limit.

The following assertions are satisfied by *SC* (proofs in [OHdG96b, SL95]) where a and b denote a client and a server endpoint, respectively:

$$\begin{aligned}
&\Box(CRQ \in C_{a,b} \wedge S.CR[b,a].Status \in \{\texttt{closed}, \texttt{open}, \texttt{closing}\} \wedge \\
&\qquad \wedge S.CR[b,a].Cache \notin \{\texttt{nil}, \texttt{old}\} \Rightarrow \\
&\Rightarrow \quad S.CR[b,a].Cache - \frac{L+W_C}{\alpha} \leq CRQ.sin \leq \\
&\qquad \leq S.CR[b,a].Cache + \frac{\max(L,W_S)+W_C+C_S}{\alpha})
\end{aligned} \tag{4.36}$$

$$\begin{aligned}
&\Box(CRQ \in C_{a,b} \wedge S.CR[b,a].Status = \texttt{opening} \Rightarrow \\
&\Rightarrow \quad S.CR[b,a].Din - \frac{L+W_C}{\alpha} \leq CRQ.sin \leq \\
&\qquad \leq S.CR[b,a].Din + \frac{L+W_C+W_S}{\alpha})
\end{aligned} \tag{4.37}$$

$$\begin{aligned}
&\Box(CRRA \in C_{a,b} \wedge S.CR[b,a].Status = \texttt{opening} \Rightarrow \\
&\Rightarrow \quad S.CR[b,a].Din - \frac{2L+W_C+W_S}{\alpha} \leq CRRA.sin \leq \\
&\qquad \leq S.CR[b,a].Din + \frac{L+W_C}{\alpha})
\end{aligned} \tag{4.38}$$

$$\begin{aligned}
&\Box(DRQ \in C_{a,b} \wedge S.CR[b,a].Status \in \{\texttt{open}, \texttt{closing}\} \Rightarrow \\
&\Rightarrow \quad S.CR[b,a].Din - \frac{L+I}{\alpha} \leq DRQ.sin \leq \\
&\qquad \leq S.CR[b,a].Din + \frac{L+W_C+I}{\alpha})
\end{aligned} \tag{4.39}$$

$$\Box(CRR \in C_{b,a} \wedge C.CR[a,b].Status \in \{\texttt{open}, \texttt{closing}\} \Rightarrow$$
$$\Rightarrow \quad C.CR[a,b].Din - \frac{L + W_S}{\alpha} \leq CRR.sin \leq \tag{4.40}$$
$$\leq C.CR[a,b].Din + \frac{L + W_C}{\alpha})$$

$$\Box(DRA \in C_{b,a} \wedge C.CR[a,b].Status = \texttt{closing} \Rightarrow \tag{4.41}$$
$$\Rightarrow \quad C.CR[a,b].Din - \frac{L + I}{\alpha} \leq DRA.sin \leq C.CR[a,b].Din)$$

$$\Box(CRRA \in C_{a,b} \Rightarrow \tag{4.42}$$
$$\Rightarrow \quad S.CR[b,a].Lin - \frac{2L + W_S}{\alpha} \leq CRRA.rin \leq S.CR[b,a].Lin)$$

$$\Box(REJ \in C_{a,b} \Rightarrow \tag{4.43}$$
$$\Rightarrow \quad S.CR[b,a].Lin - \frac{2L + W_S}{\alpha} \leq REJ.rin \leq S.CR[b,a].Lin)$$

$$\Box(CRA \in C_{b,a} \Rightarrow \tag{4.44}$$
$$\Rightarrow \quad C.CR[a,b].Lin - \frac{2L + W_C}{\alpha} \leq CRA.rin \leq C.CR[a,b].Lin)$$

$$\Box(REJ \in C_{b,a} \Rightarrow \tag{4.45}$$
$$\Rightarrow \quad C.CR[a,b].Lin - \frac{2L + I}{\alpha} \leq REJ.rin \leq C.CR[a,b].Lin)$$

$$\Box(CRR \in C_{b,a} \Rightarrow \tag{4.46}$$
$$\Rightarrow \quad C.CR[a,b].Lin - \frac{2L + W_C + W_S}{\alpha} \leq CRR.rin \leq C.CR[a,b].Lin)$$

$$\Box(DRA \in C_{b,a} \Rightarrow \tag{4.47}$$
$$\Rightarrow \quad C.CR[a,b].Lin - \frac{2L + I}{\alpha} \leq DRA.rin \leq C.CR[a,b].Lin)$$

The minimum value of N to assure the correct interpretation of modulo-N incarnation numbers can be obtained from the above assertions:

$$N \cdot \alpha \geq L + W_C + \tag{4.48}$$
$$+ \max\{L + W_C + C_S, W_C + W_S + C_S, 2L + W_C + W_S, L + 2I\}$$

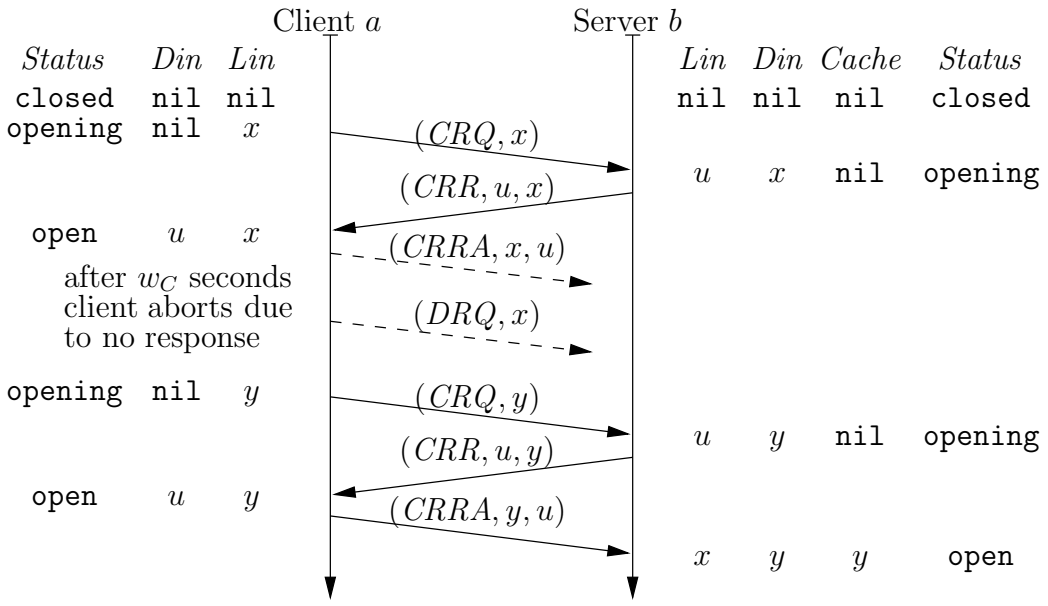


Figure 4.25: *Incarnations x and y of client a become open to incarnation u of server b.*

4.3.3 Modifications

The specification *SC* is based on the protocol presented by Shankar and Lee in [SL95], but a number of modifications have been applied to the the original specification based on the results published in [OHdG96b]. These modifications are intended to improve the behavior of the protocol. They will be explained one-by-one below.

Handling of *CRQ* messages in the `opening` state

The following code fragment is from the *RecvCR* event in Figure 4.21:

$$\textbf{elif } CR[b,a].Status = \texttt{opening} \wedge CRQ.sin > CR[b,a].Din \textbf{ then}$$
$$CR[b,a].Lin := LinGen[b]++;\ CR[b,a].Din := CRQ.sin;$$

This code fragment is executed when the server receives a recent *CRQ* message after starting a 3WHS with an old *CRQ*. The reception of the *CRQ* with the newer client incarnation number indicates that $S.CR[b,a].Din$ holds an old value, therefore a new incarnation number is generated and the new client incarnation is recorded in *Din*. In the original specification, *Din* is also updated but no new incarnation is generated.

To understand the importance of generating a new server incarnation in this case, consider Figure 4.25. The figure shows a situation when two different incarnations of client a become open to the same incarnation of server b. Of course, this must be prevented because it is a violation of the correctness requirement. The way to prevent this hazard in the original specification is to add the constraint $w_C > W_S$ to (4.35). That is, the

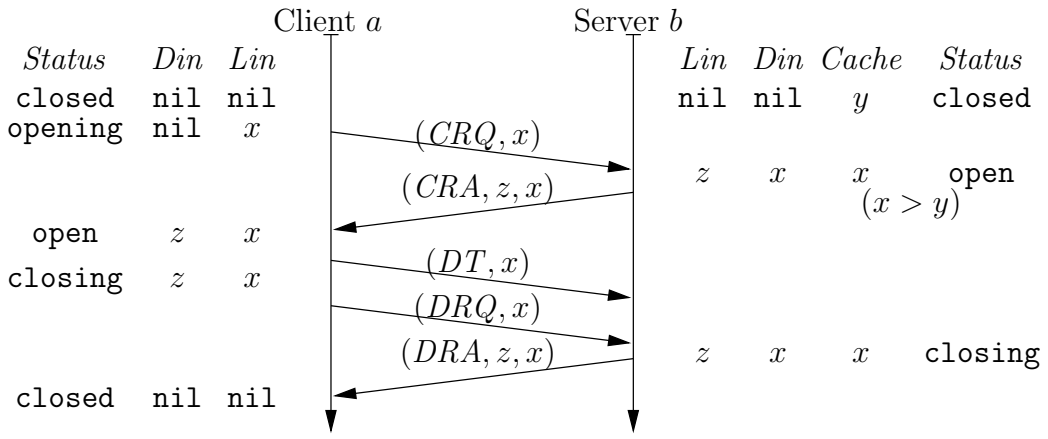


Figure 4.26: *Multi-packet transaction in the original specification.*

minimum wait time of the client in case of no response from the server must be higher than the maximum wait time of the server.

With our modification applied, there is no need for the above constraint. The situation depicted in Figure 4.25 simply cannot occur because the server always starts a new incarnation if it receives a message from a new client incarnation. In the current example, the server would respond with a (CRR, v, y) to the (CRQ, y) message. Therefore it is not possible that two client incarnations receive a message from the same server incarnation. The advantage of this modification is thus the elimination of a real-time constraint.

Early disconnect request by the client

Our specification allows the user to request a disconnect while the client incarnation is still in the `opening` state. A consequence of this behavior is that *DRQ* messages can be generated even before the client becomes open to the server. On the contrary, the original specification allows the *DiscReq* event in the `open` state only.

To keep the analysis tractable, no data transfer messages are included in the specification. The trivial way to add the data transfer function to the protocol is to define new packet types that have additional fields, such as sequence number and window size. Concerning the analysis of misinterpretable incarnation numbers, data transfer messages (*DT* and *DTA*) are equivalent to *DRQ* and *DRA* messages [Sha91, SL95].

The advantage of the 2WHS over the 3WHS is the reduced latency for transaction-oriented applications. Sending *DT* messages only after the client enters the `open` state limits these advantages to the cases when the client's request fits into the *CRQ* message. In case the request is longer, the subsequent data-transfer messages must wait until the *CRA* message arrives from the server as shown in Figure 4.26. This means a delay of one round-trip time (RTT), exactly the same delay that is imposed by the 3WHS. As the bandwidth of networks increases, the number of messages that could be sent within

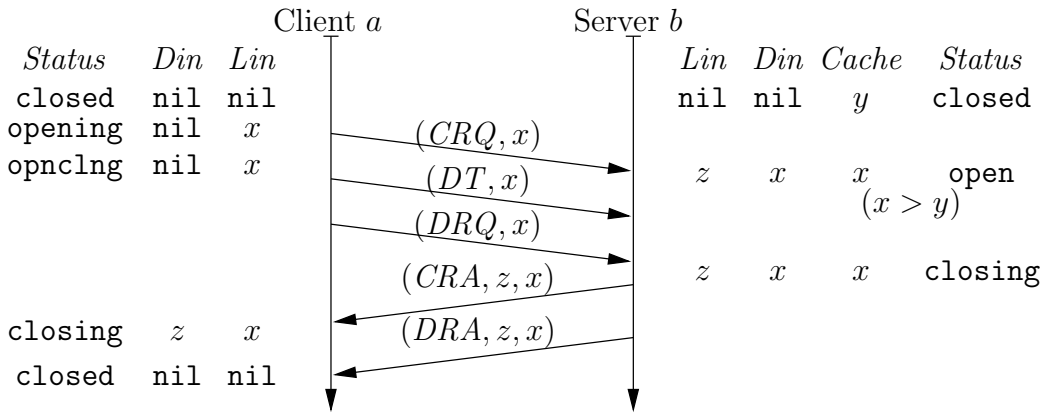


Figure 4.27: *Multi-packet transaction in SC.*

one RTT also increases making the effects of this drawback more significant.

That is why our specification allows the client to send *DRQ* and thus *DT* messages before the connection becomes open. Figure 4.27 shows the same message exchange with our protocol. Note the reduced delay of the transaction.

Our modification has a drawback, however. It raises the upper bound in assertion (4.39). The coefficient of I in constraint (4.48) becomes 2 from the original 1 due to this modification. Since I the incarnation lifetime is expected to be significantly larger than the other real-time constants in (4.48), the minimum safe value of N increases. On the other hand, a technique will be discussed below which allows to eliminate I from (4.48).

Closing handshake

The original specification instructs the server to close immediately upon the reception of a *DRQ* message. Closing means to set *Status* to `closed` and to discard the incarnation numbers stored in *Lin* and *Din*. In order to allow the client to ask for retransmissions of the *DRA* message which was sent in response to the *DRQ* message, the server responds to *DRQ* messages in the `closed` state with a *DRA* message.

This can result in the message exchange shown in Figure 4.28. There is an open but idle connection between client and server, when the server tries to transmit to the client. Because of a temporary network failure, the server receives no response and eventually aborts the connection. The same can happen if the server crashes and then recovers. The client decides to close the connection when the network is repaired. The client closes normally even though the server has aborted the connection. That is, the client is unable to distinguish between an orderly close and this situation.

In our specification, the above described situation cannot happen. A *DRA* message can only be sent while the server is in the `closing` state. After that the server responds with a *REJ* to old *DRQ* messages. Therefore, the client can be sure that the server closed

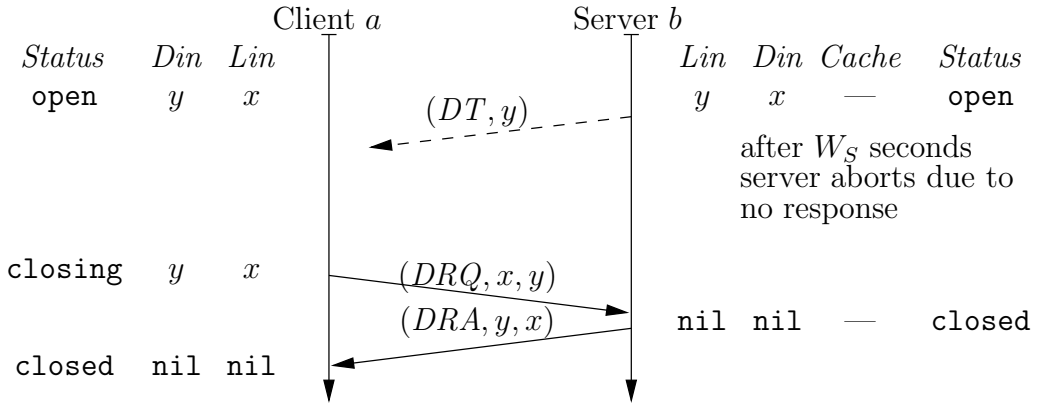


Figure 4.28: *Ambiguous closing handshake in the original protocol.*

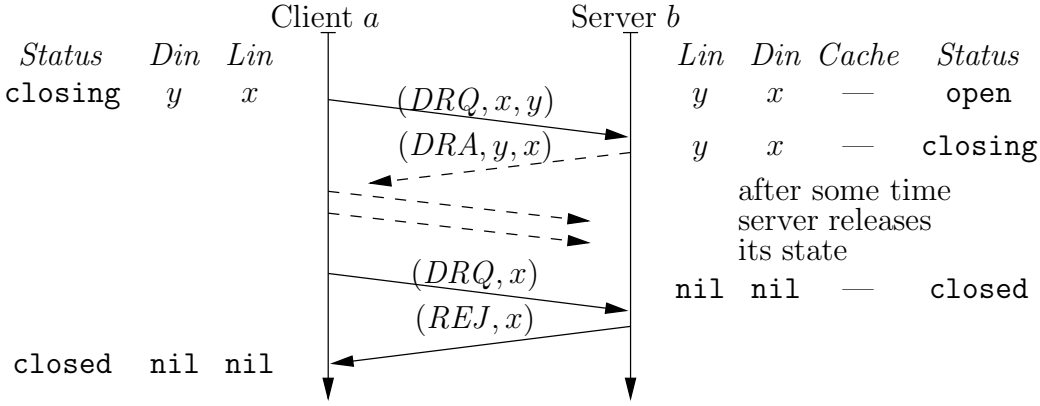


Figure 4.29: *Ambiguous closing handshake in SC.*

normally if it receives a *DRA* message. On the other hand, a different kind of ambiguity can occur. It is possible that the client receives a *REJ* message even though the server closed normally as it is depicted in Figure 4.29.

It is well-known that fully reliable close cannot be achieved over unreliable communication channels [Bel76] (see also the "two-army problem" in [Tan89]). Therefore none of the closing techniques can provide a fully reliable close because it is impossible to avoid the situation when one party has already discarded its connection state while the other is still asking for retransmissions. The difference between the two techniques is in the handling of this ambiguous case. The one used by Shankar and Lee is optimistic because it responds with a positive acknowledgment *DRA* in the lack of state information. Our proposed method is pessimistic because it responds with a negative acknowledgment *REJ* is those situations. One advantage of our method is that it allows to lower the bound in (4.41).

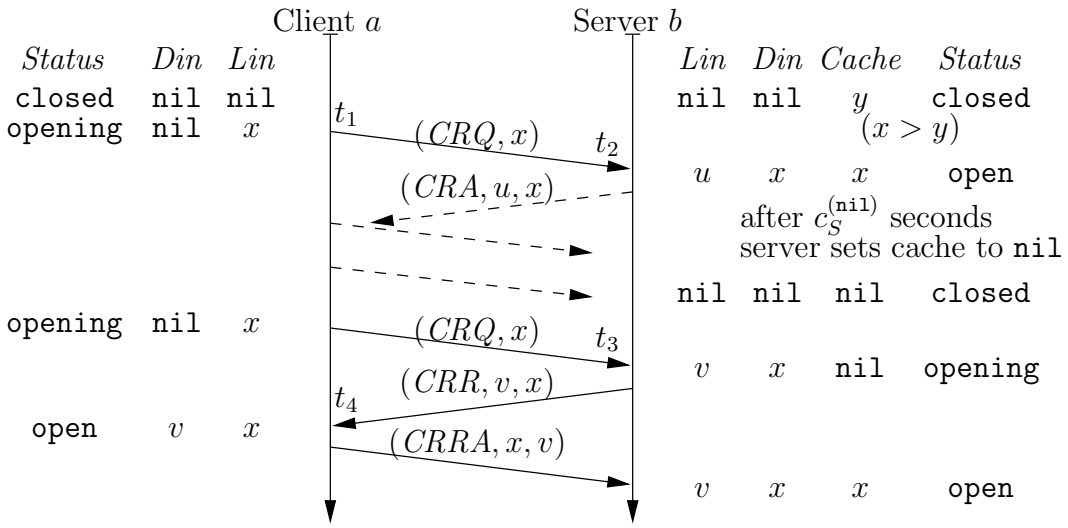


Figure 4.30: *Why must* $c_S^{(\texttt{nil})} > W_C$ *hold?*

4.3.4 Reducing the necessary caching period

One important feature of connection management protocols is the size of the cache needed. In case of big servers, this can be a considerable amount of memory. The necessary cache size is determined by the size of a cache entry and the caching period. In *SC* a cache entry consists of a single identifier bundled with some control information such as the associated endpoint addresses and the cache residency time.

The necessary caching periods ($c_S^{(\texttt{nil})}$ and $c_S^{(\texttt{old})}$) are determined by (4.35). Our goal is to reduce these bounds. Although these bounds are tight in the sense that for any bound lower than these it is possible to create a protocol sequence which violates the correctness properties, we will show that the caching period can be reduced in common situations.

The proposal can be best explained through the cases where the conditions of (4.35) are indeed necessary. Figure 4.30 shows a situation when the violation of $c_S^{(\texttt{nil})} > W_C$ can result in two server incarnations becoming connected to the same client incarnation. From the protocol specification, $t_4 - t_1 < W_C$ must hold because the client did not abort incarnation x. The *CRR* message from the server incarnation v must arrive before the client aborts incarnation x, otherwise the 3WHS never completes. On the other hand, $t_3 - t_2 > c_S^{(\texttt{nil})}$ since x is replaced by `nil` in the cache. Combining these with the obvious $t_1 < t_2$ and $t_3 < t_4$ yields the first part of (4.35).

Figure 4.31 shows a similar case to explain the bound on $c_S^{(\texttt{old})}$. Here the second incarnation of the server also connects by 2WHS. The longest possible gap between two *CRQ* messages from the same client incarnation is $W_C + L$, therefore $c_S^{(\texttt{old})} > W_C + L$ is necessary to avoid the sequence in Figure 4.31.

From these examples we can formulate the general rule for discarding a cached entry. A cache entry x can be set

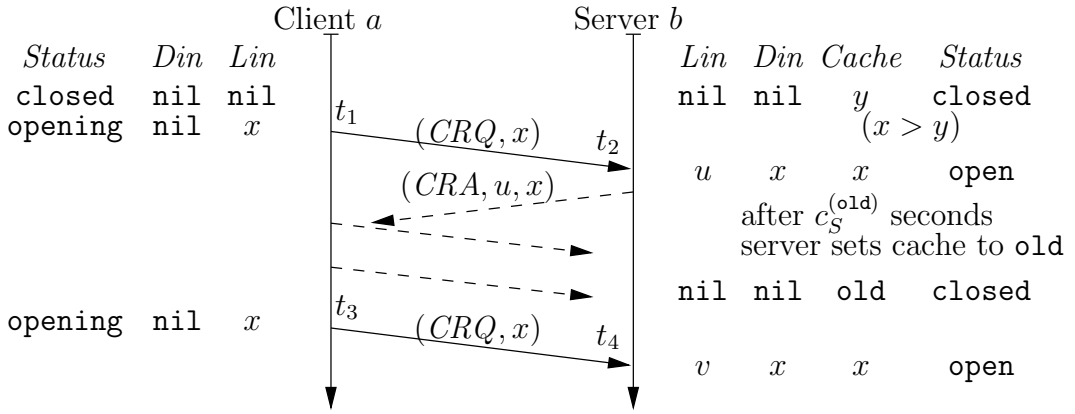


Figure 4.31: *Why must* $c_S^{(\texttt{old})} > W_C + L$ *hold?*

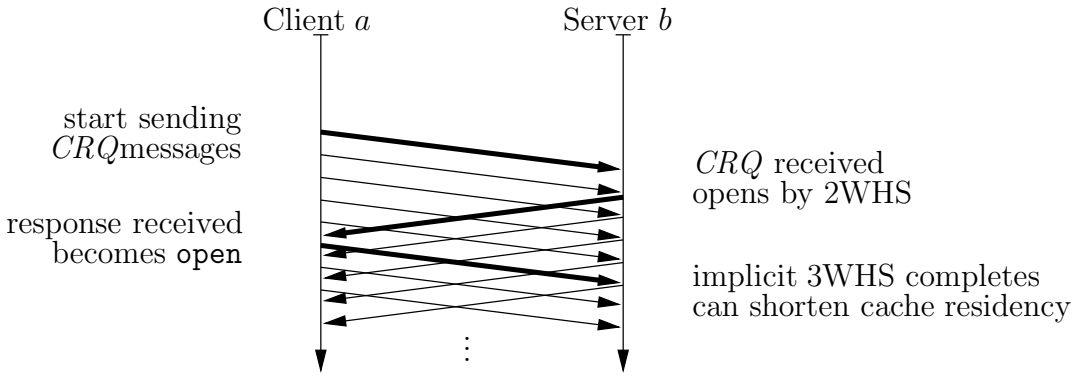


Figure 4.32: *The implicit 3WHS.*

- to `nil` when the server cannot complete a 3WHS with incarnation x of the client any more;
- to `old` when no more (CRQ, x) messages exist or can be generated.

The 3WHS can be completed until the client is in the `opening` state[4]. The upper bound on the lifetime of a CRQ message from that incarnation is L seconds after it left the `opening` state.

In the above examples, the server's estimate of the upper bound of incarnation x being in the `opening` state is $t + W_C$, where the first (CRQ, x) message is received at time t by the server. Another indication of the client being in a state other than `opening` is the completion of the 3WHS. When the server receives a $CRRA$ message, it can be sure that the client is not in the `opening` state any more. Therefore, the server can set the cached entry to `nil` at any later time without enforcing the $c_S^{(\texttt{nil})}$ minimum caching period. Furthermore, the cache can be set to `old` L seconds after the reception of the $CRRA$.

This optimization is not limited to the cases when the 3WHS is used to open the con-

[4]The `opnclng` state is equivalent to `opening` in this respect.

nection. The caching period can be shortened in the same way when the server is sure that the client is in the `open` state or beyond. A clear indication of this is when the server receives a message from the client which is a reply to an earlier server message. This is shown in Figure 4.32. The sequence is like a 3WHS, but the data transfer can start before the handshake completes. It will be called *implicit 3WHS*.

Therefore the required cache residency time can be reduced when the 3WHS, either explicit or implicit, completes. The optimization can result in a considerable reduction of the cache size, since during normal operation the majority of connection setup requests succeeds. Furthermore, W_C is likely to be much longer than L, thus the bigger part of the compulsory cache residency time is eliminated. The only cases when the server has to cache an entry for the full period required by (4.35) are when the server opens with 2WHS and

- the client never becomes open due to a transient network failure as shown in Figure 4.30 and 4.31;
- the communication is a short transaction and all the client data is carried in the first few packets which are sent before the server's *CRA* arrives.

The first case occurs only rarely and the second case can be avoided by modifying the specification of the client such that it acknowledges the *CRA* message if there are no more data messages to be sent when the *CRA* arrives.

4.3.5 TCP for Transactions

The practical importance of the protocol *SC* is shown by the fact that a recently proposed extension to TCP is based on very similar techniques. TCP for Transactions (T/TCP) is specified in [Bra92, Bra94] and it has already been implemented in some operating systems [Ste96]. It is instructive to examine how the techniques of the abstract specification *SC* appear in the actual transport protocol, T/TCP. To facilitate this analysis, the most important features of T/TCP are captured in the specification *T/TCP* in Figure 4.33. This specification is obtained by modifying the specification *SC* where it is necessary.

The most important difference between *T/TCP* and *SC* is that *T/TCP* allocates only one cache entry for each remote host. In the specification *SC*, cache entries were assigned to pairs of endpoint addresses. If a_1 and a_2 are different client endpoint addresses, then the server endpoint b has two separate entries, $CR[b, a_1].Cache$ and $CR[b, a_2].Cache$, regardless of the fact whether a_1 and a_2 belong to the same host or not. In case of *T/TCP*, these connections share the same *Cache* entry if a_1 and a_2 belong to the same host.

The definition of host and endpoint addresses was discussed in Section 4.1. A host address in the formal model is equivalent to an IP address in TCP/IP terminology. An endpoint address of the formal model is equivalent to an ⟨IP address,TCP port⟩ tuple. The function $host(a)$ gives back the host address of the endpoint a.

program *T/TCP*;
type
host_addr = ⟨type of host addresses⟩;
ep_addr = ⟨type of endpoint addresses⟩;
clnt_conn_rec = **record of**
Status: {`closed`, `opening`, `opnclng`, `open`, `closing`};
Lin, *Din*: {`nil`} ∪ {0, 1, ...};
srvr_conn_rec = **record of**
Status: {`closed`, `opening`, `open`, `closing`};
Lin, *Din*: {`nil`} ∪ {0, 1, ...};
⟨Packet types are not affected.⟩
channel ⟨one channel $C_{a,b}$ for each a, b pair of endpoints⟩;

process *C*;
var *CR*: **array** [*ep_addr*, *ep_addr*] **of** *clnt_conn_rec*;
LinGen: {0, 1, ...};
event *ConnReq*(*a*, *b*);
when *CR*[*a*, *b*].*Status* = `closed` **then**
CR[*a*, *b*].*Status* := `opening`; *CR*[*a*, *b*].*Lin* := *LinGen* + +;
⟨No other client events are affected by the modification.⟩;

process *S*;
var *CR*: **array** [*ep_addr*, *ep_addr*] **of** *srvr_conn_rec*;
Cache: **array** [*host_addr*] **of** {`old`, `nil`} ∪ {0, 1, ...};
LinGen: {0, 1, ...};
Modified server events are specified in Figure 4.34 and 4.35;

Figure 4.33: *T/TCP: main program*

Allocating only one cache entry per host is a way to reduce the amount of memory used by the protocol. The saving can be significant when a client host opens many connections from different endpoint addresses to the same server within a short period. This is exactly what happens in most TCP implementations which assign successive endpoint addresses (port numbers in TCP terminology) to connections opened by a user in a series. Such a sequence of connections occupies n different cache entries in case of *SC*, but only one in case *T/TCP*.

Because of the single cache entry per remote host, *T/TCP* uses a single incarnation number generator *LinGen* per host. This is necessary because all incarnation numbers created at the client host *C* are stored in the same cache entry by the server. The monotonicity of these client incarnation numbers can only be guaranteed if they are generated by a common source at host *C*.

The client specification needs no further changes besides the definition of *LinGen*. The

event *MakeNil*(b, a);
  when *Cache*$[host(a)] \notin \{\texttt{nil}, \texttt{old}\} \wedge$
    ⟨entry was updated between $c_S^{(\texttt{nil})}$ and $c_S^{(\texttt{old})}$ seconds ago⟩ **then**
    Cache$[host(a)] := \texttt{nil}$;
event *MakeOld*(b, a);
  when *Cache*$[host(a)] \notin \{\texttt{nil}, \texttt{old}\} \wedge$
    ⟨entry was updated between $c_S^{(\texttt{old})}$ and C_S seconds ago⟩ **then**
    Cache$[host(a)] := \texttt{old}$;
event *RecvCRRA*(b, a);
  when head$(C_{a,b}) = CRRA$ **then** $CRRA := \textbf{recv}(C_{a,b})$;
    if $CR[b, a].Status = \texttt{opening} \wedge CRRA.sin = CR[b, a].Din \wedge$
      $CRRA.rin = CR[b, a].Lin$ **then**
      $CR[b, a].Status := \texttt{open}$;
      if *Cache*$[host(a)] \in \{\texttt{nil}, \texttt{old}\} \vee CRRA.sin > Cache[host(a)]$ **then**
        $Cache[host(a)] := CRRA.sin$;
    elif $CR[b, a].Status \in \{\texttt{open}, \texttt{closing}, \texttt{closed}\}$ **then** ⟨no action⟩;
⟨The events *Release*, *Abort*, *SendCRR*, *RecvDR*, and *RecvREJ* are not affected.⟩;

Figure 4.34: *T/TCP: modified server events, part 1*

modifications to the server events are shown in Figure 4.34 and 4.35. The most important changes are related to the way a 3WHS is done:

- In the event *RecvCR*, a 3WHS is started when $CRQ.sin < Cache[host(a)]$. In the specification *SC*, a 3WHS is started only when the cache is `nil`; the *CRQ* message is discarded when *sin* is less than the cached value.
- In the event *RecvCRRA*, the cache cannot be updated unconditionally with the client's incarnation number *CRRA.sin*. A check must be made first to see if *CRRA.sin* is greater than the value currently in the cache.

These modifications are necessary because the situation in Figure 4.36 is possible in *T/TCP*. Assume that a_1 and a_2 are endpoint addresses belonging to the client host C, and b is an endpoint address of server S. Assume further that both of these endpoints become `opening` to b, i.e., $CR[a_1, b].Status = CR[a_2, b].Status = \texttt{opening}$ and $CR[a_1, b].Lin = x_1 < CR[a_2, b].Lin = x_2$. It is possible that b becomes `open` to a_2 first even though incarnation x_1 of a_1 started earlier than incarnation x_2 of a_2. In that case, the server has x_2 ($> x_1$) in its cache when the valid request (CRQ, x_1) of a_1 arrives. Therefore, the server engages in a 3WHS with the client if $CRQ.sin$ is less than the cached incarnation number in order not to reject possibly valid *CRQ* messages.

The other modification in the event *RecvCRRA* is also related to the situation depicted in Figure 4.36. Assume that the 3WHS with incarnation x_1 of a_1 completes normally, as it is shown in the figure. In *SC*, the client incarnation number was simply written to the cache at this point. In case of *T/TCP*, it must be checked first if the client's incarnation number is greater than the value in the cache. It is important that the cached value is

```
event RecvCR(b, a);
    when head(C_{a,b}) = CRQ then CRQ := recv(C_{a,b});
        if CR[b, a].Status = closed ∧ ⟨rejecting connections⟩ then
            REJ.rin := CRQ.sin; send(C_{b,a}, REJ);
            if Cache[host(a)] = old ∨ CRQ.sin > Cache[host(a)] ≠ nil then
                Cache[host(a)] := CRQ.sin;
        elif CR[b, a].Status = closed ∧ ⟨accepting connections⟩ ∧
                (Cache[host(a)] = nil ∨ CRQ.sin < Cache[host(a)] then
            CR[b, a].Status := opening; CR[b, a].Lin := LinGen + +;
            CR[b, a].Din := CRQ.sin;
        elif CR[b, a].Status ∈ {closing, closed} ∧ ⟨accepting connections⟩ ∧
                (Cache[host(a)] = old ∨ CRQ.sin > Cache[host(a)] ≠ nil) then
            CR[b, a].Status := open; CR[b, a].Lin := LinGen + +;
            CR[b, a].Din := CRQ.sin; Cache[host(a)] := CRQ.sin;
            CRA.sin := CR[b, a].Lin; CRA.rin := CR[b, a].Din; send(C_{b,a}, CRA);
        elif CR[b, a].Status = opening ∧ CRQ.sin > CR[b, a].Din then
            CR[b, a].Lin := LinGen + +; CR[b, a].Din := CRQ.sin;
        elif CR[b, a].Status = open ∧
                (Cache[host(a)] = old ∨ CRQ.sin > Cache[host(a)] ≠ nil) then
            if ⟨willing to reopen⟩ then
                CR[b, a].Lin := LinGen + +; CR[b, a].Din := CRQ.sin;
                Cache[host(a)] := CRQ.sin;
                CRA.sin := CR[b, a].Lin; CRA.rin := CR[b, a].Din; send(C_{b,a}, CRA);
            else
                CR[b, a].Status := closed; CR[b, a].Lin := nil; CR[b, a].Din := nil;
                Cache[host(a)] := CRQ.sin;
        elif CR[b, a].Status = open ∧ CRQ.sin = Cache[host(a)] ∉ {nil, old} then
            CRA.sin := CR[b, a].Lin; CRA.rin := CR[b, a].Din; send(C_{b,a}, CRA);
```

Figure 4.35: *T/TCP: modified server events, part 2*

not lowered, because if x_1 were written to the cache, then a duplicate of (CRQ, x_2) could be erroneously accepted later on.

The difference between *SC* and *T/TCP* can be explained in another way which will help us to find the correctness conditions of *T/TCP*. In case of *SC*, the validation of *CRQ* messages has three possible outcomes:

- *CRQ* is *known to be new* when $CRQ.sin > Cache$. In this case a 2WHS is sufficient to open the connection.
- *CRQ may be new* when *Cache* contains `nil`. In this case a 3WHS is necessary to open the connection.
- *CRQ* is *known to be old* when $CRQ.sin < Cache$. In this case the request is ignored or rejected.

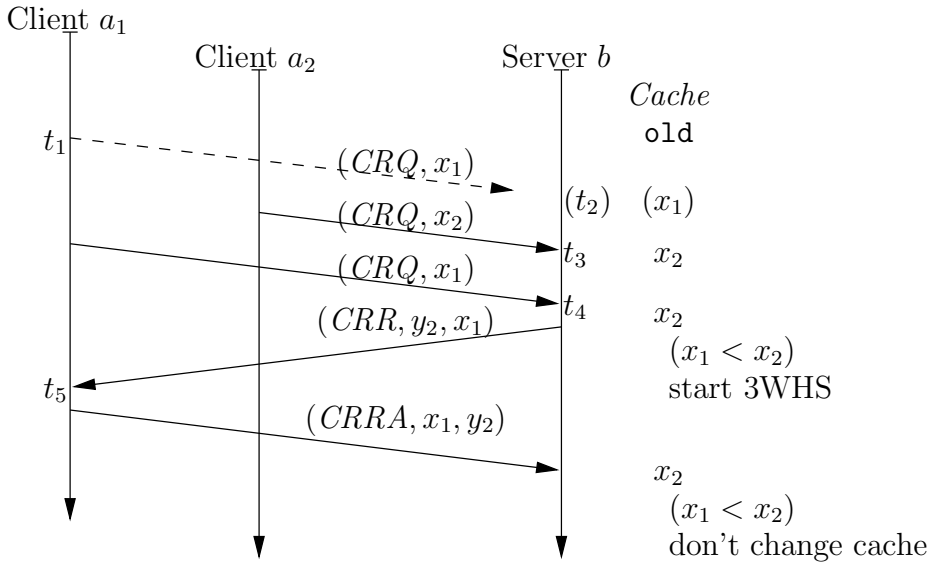


Figure 4.36: *Two client endpoints establishing connections to a server.*

In case of T/TCP, the third case is not applicable. A CRQ message may be new even if $CRQ.sin < Cache$ as it was shown above. Therefore in T/TCP, a 3WHS is always possible unlike in SC which has situations when neither a 2WHS nor a 3WHS can be used to open a connection.

This observation is important for the correctness of the protocol. To see how a protocol error shown in Figure 4.30 can occur with T/TCP, consider Figure 4.36 once more. Assume that the first CRQ message from incarnation x_1 is not lost. In this case the connection is opened by a 2WHS at time t_2 because the cache contains the value `old`. Let the server create incarnation y_1 at t_2 while processing the (CRQ, x_1) message. The CRQ message of incarnation x_2 is accepted at t_3. A retransmission of (CRQ, x_1) is received at t_4. Assume that the server aborts incarnation y_1 somewhere between t_2 and t_4 due to no response from the client. Therefore at t_4 the server starts a new incarnation y_2 in response to the retransmitted (CRQ, x_1) message. If the reply, the (CRR, y_2, x_1) message reaches the client before it aborts incarnation x_1, then the 3WHS can succeed and the server can become connected to the same client incarnation twice.

The protocol error above can be avoided by the following constraint:

$$w_S > W_C \tag{4.49}$$

where w_S is the lower bound of the server's waiting period. From the protocol specification $t_4 - t_2 > w_S$ and $t_5 - t_1 < W_C$ follow, therefore (4.49) assures that the 3WHS cannot be completed any more when the server aborts its incarnation due to no response from the client. (4.49) is an additional term to the conditions in (4.35) which is needed for T/TCP only due to its different caching scheme.

4.4 Discussion of CM protocols

Several different CM protocols have been analyzed so far. The purpose of this section is to discuss some issues that are common to all of them and to compare the protocols.

4.4.1 State management

Each CM protocol stores state information about the active connections in connection records (CR). The SC family of protocols, both the specification *SC* and *T/TCP*, use a cache as well. The cache can be seen as a reduced CR which contains essential information about recent connections.

In the formal model, there is a CR for each possible connection and that makes the formal verification simpler. There is no need to create a CR for every potential connection in an implementation, however. Actually, it is not even feasible due to the large number of potential connections. An implementation maintains a CR for a connection only if it is not in the `closed` (idle) state. The lack of a CR for a connection implies that the connection is in the `closed` state.

Cache management policies in *SC* and *T/TCP*

Similar techniques can be applied to implement the *Cache* in the *SC* and *T/TCP* protocols. Each entry in the cache can store an incarnation number, or have the special values `nil` or `old`. Similarly, to CRs, it is not feasible to store every logical cache entry of the formal specification in the physical cache maintained by an implementation. The actual cache management policy, which depends on how the events *MakeNil* and *MakeOld* are invoked, determines the correspondence between logical and physical cache entries.

Different cache management techniques are discussed in [SL95]. One possibility is when `nil` entries are not stored in the physical cache. That is, no entry in the physical cache for a connection means that the corresponding logical cache entry is `nil`. The disadvantage of this technique is that the first connection between a client and server is always opened by the 3WHS.

An alternative cache management policy is when no entries in the cache are ever set to `nil`. That is, the event *MakeNil* is never invoked. This means that each entry in the cache must be retained for at least $c_S^{(\texttt{old})}$ seconds. On the other hand, in this case the lack of an entry in the physical cache can be defined to mean that the corresponding logical cache entry is `old`. The advantage is thus that *CRQ* messages from a client which has no corresponding entry in the physical cache of the server are always accepted by the 2WHS.

It is also possible to combine these two cache management policies. The implementation can maintain a flag which indicates which interpretation of the physical cache is in effect. By default, the lack of an entry in the physical cache means that the logical entry is `old`. If, however, an entry from the physical cache must be discarded earlier than $c_S^{(\texttt{old})}$, then the interpretation changes so that no entry in the physical cache means `nil`. The interpretation can be changed back to "no entry means `old`" if no cache entries are discarded from the physical cache for a period of $c_S^{(\texttt{old})}$. This combined management policy allows to determine the trade-off between memory usage and connection-setup latency dynamically.

State management in *SCMP*

For the first sight, the techniques used in *SCMP* (timestamps and synchronized clocks) look quite different from the caching techniques used in the SC family of protocols. In fact, the differences are not substantial as the following analysis reveals.

Assume an implementation of *SC* which uses a clock in each host to implement *LinGen* and which uses the second cache management policy ("no entry means `old`"). If the *LinGen* clocks are synchronized, then the non-`old` entries in the cache are within close bounds of each other.

The basic difference between *SCMP* and the above described version of *SC* is that *SCMP* exploits the fact that incarnation numbers are from synchronized sources in its hypothetical cache management policy. Assume that the physical cache of the *SC* implementation is full of non-`old` entries. The server is not allowed to discard these entries before their $c_S^{(\texttt{old})}$-timer expires in order to prevent the protocol error shown in Figure 4.31.

The protocol *SCMP* maintains the upper bound of the timestamps (incarnation numbers in *SC*) from the discarded CRs in *upper*. Discarding a CR in *SCMP* is equivalent to discarding a cache entry in *SC* in this context. In the protocol, *upper* acts as a collective entry in the cache for all the entries that were discarded. This assures that no duplicate connection request are accepted (example in Figure 4.31) even if the entries are discarded earlier than the compulsory caching period $c_S^{(\texttt{old})}$ in case of *SC*.

On the other hand, *upper* is only an estimate of the discarded timestamps. Therefore, it is possible that a valid request is rejected even if its timestamp is larger than the last timestamp used on that connection. This happens if the CR of the connection has already been discarded and *upper* happens to be larger the the timestamp of the new request. The probability of such cases depends on the quality of the clock synchronization.

Therefore, clock synchronization allows to reduce the caching period in *SCMP* with respect to the caching period in *SC*. The drawback is that valid requests may be rejected if the clock synchronization assumption is violated. All other differences between *SCMP* and *SC* are marginal.

4.4.2 Incarnation lifetime

The correctness of *SC* and *T/TCP* depends explicitly on the incarnation lifetime I when modulo-N identifiers are used (see (4.48)). In the specification *SCMP*, there is no explicit parameter to denote the maximum incarnation lifetime, but the event *ClkTick* aborts connections that are in a state other than `closed` longer than a certain period (see Figure 4.4 and 4.5). This is an implicit bound on incarnation lifetimes.

The dependency on the incarnation lifetime is problematic. The lifetime of a connection is determined by the user of the transport service and the actual connection lifetime varies strongly from application to application. Enforcing an upper bound on I is not a good solution, it would be much better to remove I from the condition for the correctness of the protocol.

The following example shows how a long-lived incarnation can cause a protocol error. The example uses *SC*, but the same argument holds for *SCMP* and *T/TCP*. Assume that a long-lived connection is immediately followed by another connection between the same client-server pair. If the connection lasted long enough, then it is possible that the new connection receives the same incarnation number x because *LinGen* wrapped around. The problem with this is that *DRQ* and data packets of the old connection can be erroneously accepted in the new connection because they carry the same modulo-N incarnation number.

Note that the error could occur because the uniqueness of the incarnation number assigned to the new connection is not guaranteed if the previous incarnation uses an incarnation number too long. Limiting I is one way to eliminate this hazard because it provides an explicit bound on the age of incarnation numbers.

Another technique called *Lin*-generator cache, is proposed in [SL95]. The idea is to assure that successive connection attempts are identified with "close-by" incarnation numbers. The *Lin*-generator cache stores the latest incarnation number of each connection for a period of $2L$. When an entity with a *Lin*-generator cache becomes involved in a connection attempt with a remote entity, it obtains its local incarnation number as follows: if its *Lin*-generator cache contains an entry for the remote entity, it uses an incarnation number one higher than the entry; otherwise, it uses an arbitrary incarnation number.

This technique restores the uniqueness of incarnation numbers assigned to new connection attempts by remembering the last used incarnation number on that connection until all messages with the old incarnation number disappear from the network. The disadvantage is that it requires an additional cache.

We propose an alternative method without additional caching. If the incarnation numbers carried in the *DRQ* and data packets are always taken from the current value of *LinGen*, then we can always generate unique incarnation numbers from *LinGen* when a new connection attempt is made. This is implied by the monotonicity of *LinGen* and the

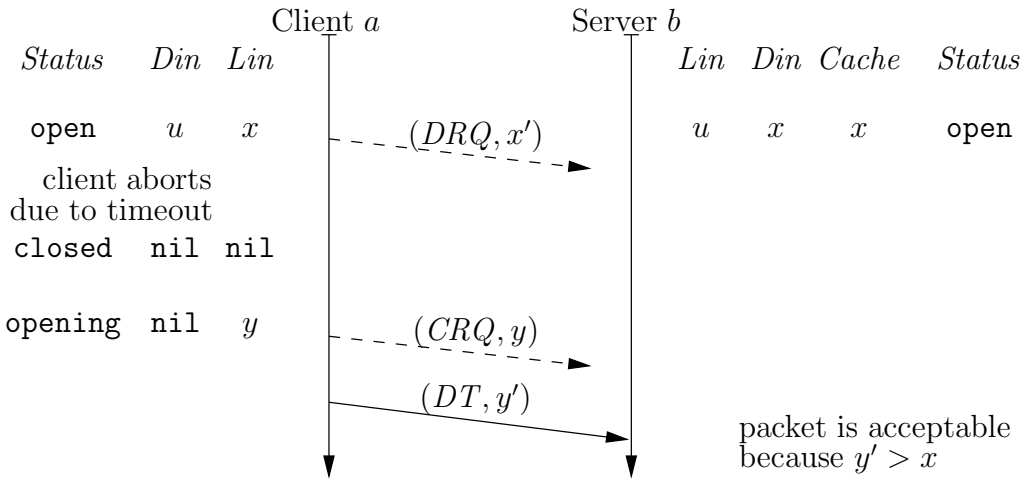


Figure 4.37: *Accepting a new DT packet as part of an old connection.*

limited packet lifetime. The server accepts such a data packet if its incarnation number is not lower than the most recent incarnation number that was received on the same connection. The technique is identical to the technique used to check the timestamp in data packets of PAWS in Section 3.3.

It is crucial that all *CRQ*, *CRR*, and *CRA* messages of a connection get their incarnation numbers assigned in the original method defined in *SC*. That is, the number is taken from *LinGen* when the incarnation is started and all retransmissions of the message carry the same incarnation number.

Furthermore, data and *DRQ* messages sent in the `opening` and `opnclng` states must be treated specially. These messages can be erroneously accepted by an old server incarnation if the *DRQ* message of the old connection and the *CRQ* message of the new connection is lost. The situation that is shown in Figure 4.37, can happen because in the modified protocol the server accepts a *DRQ* (or *DT*) message if its incarnation number is higher than the highest incarnation number received so far. From the figure we can see that $x \leq x' < y \leq y'$ holds.

To avoid the above error, the client is required to send a *CRQ* message along with *DT* and *DRQ* messages that are sent in the `opening` and `opnclng` states. In these states, the client does not know if the server has already seen the *CRQ* message of the new connection attempt. The server processes these compound messages if a *CRQ* message were immediately followed by the *DT* or *DRQ* message. Processing the *CRQ* first assures that the server recognizes the opening of the new connection, therefore the *DT*/*DRQ* message cannot be delivered to the old incarnation.

If these modifications are applied to the protocol *SC*, for example, then the real-time conditions for the correctness of the protocol change as described below. The changes to the other protocols *SCMP* and *T*/*TCP* are similar. The client is allowed to a keep

a connection incarnation indefinitely once it has entered the `open`[5] state. The upper bound W_C on the length of the `opening` state is still necessary.

The server is allowed to keep a connection incarnation indefinitely when it has completed a 3WHS with the client, either explicit or implicit. The upper bound W_S of completing the 3WHS remains in effect. When the server completes the 3WHS, i.e., it receives an ack of its *CRR* or *CRA*, then it is sure that the client is not retransmitting the *CRQ* message. All the subsequent *DT*, *DTA*, *DRQ*, and *DRA* messages are validated using the technique for validating the timestamps in PAWS and no limits on the length of a data transfer session are necessary to guarantee the correct interpretation of these timestamps.

An advantage of the technique is that it does not affect the crash-recovery mechanisms of the CM protocol. This is so because the modified protocol draws the incarnation numbers from the same incarnation number generator as the original connection management protocol to which the modification is applied. Any connection setup protocol must assure that the incarnation numbers used after recovery from a crash could not have been used before the crash. Since the packets of the modified protocol use the same incarnation numbers, pre-crash packets can never be mixed with post-crash packets so the modified protocol will work as expected.

Notice that the incarnation numbers in *DT*, *DTA*, *DRQ*, and *DRA* messages can be eliminated if PAWS is used for the data-transfer part of a transport protocol. To do this, the same clock must be used to generate the incarnation identifiers of the CM protocol and the timestamps of PAWS. The incarnation number *CRQ.sin* behaves like an initial timestamp for PAWS, because it is used at the server to initialize the PAWS variable *R.TsRec*. If data transfer is to be established in the reverse direction then *CRA.sin* can used to initialize *R.TsRec* at the client side. SCMP could be used for the connection management part in the same way.

A technique is proposed in [Sha91] to combine connection management and data transfer protocols systematically. The technique keeps the identifier spaces of the connection management and data transfer protocols separate. Our technique works by merging the identifier spaces of the connection management and data transfer protocols. The advantages of our technique are that (i) it eliminates the dependency on the incarnation lifetime and (ii) it reduces the protocol header overhead. On the other hand, the technique in [Sha91] can be applied to any pair of CM and data transfer protocols, while our technique works with specific protocols only.

[5] Or `closing` if `open` was skipped due to an early disconnect request

Chapter 5

Conclusion

This thesis is devoted to the design and analysis of reliable transport protocol mechanisms. Data transfer and connection management protocols belong to this area. Several such protocols have been proposed since the inception of computer networking. The design of these protocols is heavily influenced by the current networking technology. This influence appears in, sometimes implicit, assumptions about

- the properties of the network-level service, such as packet lifetimes, and the presence of reordering and/or duplication in the network;
- the relative cost of computing (CPU cycles) versus network bandwidth;
- the existence of auxiliary services such as clock synchronization.

The increased research and development activities in transport protocol design around the late eighties, early nineties can be attributed to the influence of advances in high-speed networking technologies such as ATM and improvements in the TCP/IP protocol suite.

The goal of the research which lead to this thesis was to increase the understanding of these protocol mechanisms by systematic analysis and comparison. We believe that many of the different protocols proposed in the literature can be traced back to a few fundamental protocol mechanisms. Although the definition of the basic mechanisms from which all reliable transport protocols can be derived is not presented here, we believe that the discussion of many protocol design issues throughout the thesis takes us one step closer to the ultimate goal.

Another important goal which was defined at the very beginning of the research is that we always take into account the effect of using a bounded identifier space in our analyses. It may sound obvious that all identifiers in protocols must be from a modulo-N space for some N, but still many protocol verifications in the literature assume that identifiers are unbounded integers. Certainly, in some situations one can argue that N is very large and can be assumed unbounded. But in many cases such an assumption is likely to be invalid, for example in very high-speed networks where the identifier space is consumed relatively fast. Therefore we strongly believe that understanding the effects of using

modulo-N identifiers in transport protocols is crucial to the thorough understanding of these protocols.

5.1 Contributions

We have contributed to the understanding of protocol design in two ways:

- by the formal verification of some protocols, PAWS in Section 3.3 and SCMP in Section 4.2;
- by informally discussing several other protocol design issues, such as the data transfer protocol SNR in Section 3.4, the connection management protocols SC and T/TCP in Section 4.3, and issues regarding the way connection management and data transfer protocols are combined into a full-featured transport protocol in Section 4.4.2.

The data transfer protocol, PAWS has been verified in Section 3.3. PAWS [JBB92] is an interesting technique to extend the sequence number space of sliding-window protocols and at the same time to add extra functionality, namely simplified round-trip time measurements. The formal verification has revealed some problems with the protocol correctness which have been corrected. Furthermore it provided the exact terms under which the protocol can be operated safely. The comparison with the plain sliding-window protocol has shown that the design of PAWS is a trade-off between the extra functionality and protocol implementation constraints such as lower allowed transmission rate and more restricted liveness assumptions.

Another data transfer protocol was analyzed in Section 3.4. SNR [NRS90, GNS95] presents a different trade-off between the transmission rate and restrictions on the network and protocol behavior. SNR allows a higher maximum transmission rate at a given N than the plain sliding-window protocol by assuming that neither the network nor the transport protocol creates duplicates. The assumption of no duplicates allows SNR to use the sequence number space more efficiently; in SNR the maximum window size is equal to the whole sequence space, while in the plain sliding-window protocol it can be at most half of the sequence number space. On the other hand, the "no duplicates" assumption creates a coupling between protocol correctness and performance because the retransmission policy, which is normally influenced by performance considerations only, must assure that no duplicates are created. This lack of orthogonality between protocol correctness and performance can be costly in certain networking environments, such as the Internet. An alternative specification of SNR has also been presented which captured the idea of periodic state exchange, but kept the performance and protocol correctness issues independent.

The novel idea in the design of SCMP [LSW91] is the use of network-wide synchronized clocks and timestamps. The formal verification of SCMP is presented in Section 4.2. Our verification has proven that SCMP satisfies the desired safety properties of the connection management service in [Sha91]. The novelty of our verification is the proof

of the conjecture that only minimal assumptions about the network are sufficient for the safety of the protocol. In particular, clock synchronization is not necessary for safety; even if modulo-N timestamps are used, bounded clock drift is sufficient. Obviously, the lack of clock synchronization adversely affects the performance of the protocol because valid connection requests can be rejected.

So-called cache based connection management protocols are analyzed in Section 4.3. The common feature of these protocols is that they use the memory for protocol state information as a cache. If the necessary information is present in the cache then connections are opened by the low latency 2WHS, otherwise the 3WHS must be used. Such a cache based protocol SC was described in [SL95]. Modifications to the original SC specification have been presented in the thesis which allow to eliminate some real-time conditions for the correctness of the protocol and to reduce the latency of multi-packet transactions. Some issues regarding the graceful close of a transport connection have also been analyzed. Furthermore, the concept of implicit 3WHS have been introduced and it has been shown how it can be exploited to reduce the required caching period in common situations.

The formal model of T/TCP [Bra92, Bra94] has been derived from the specification of SC. T/TCP is an extension to the standard transport protocol of the Internet which provides TCP with the option of opening connections with the 2WHS. It has been shown that the basic difference between SC and T/TCP is in the granularity of caching. SC assigns a (logical) cache entry to each remote transport endpoint while T/TCP assigns cache entries to remote hosts. The effects of this difference on the other protocol mechanisms have been described and analyzed.

The relation between SCMP and the SC family of protocols has been described in Section 4.4. The outcome of our analysis is that SCMP can be considered as a specialization of the SC protocols which uses a very specific cache management technique based on the assumption of synchronized clocks. The use of clock synchronization allows SCMP to use its cache more efficiently.

We have also shown that the correctness of connection management protocols can be assured without assuming a maximum incarnation lifetime. This result is practically important because the incarnation lifetime is determined by applications and it is very difficult to enforce an upper bound on incarnation lifetimes. This result contradicts the claim in the literature that dependency on the maximum incarnation lifetime is mandatory when modulo-N identifiers are used. Our solution can be easily implemented when the identifier spaces of the connection management and data transfer protocols are shared. It has been argued informally that the usage of this technique does not affect the crash-recovery behavior of the protocol.

5.2 Remaining issues

No *progress properties* of the connection management protocols have been proven in Chapter 4. Progress of the connection management service could be defined such that if a is opening to b, then eventually a becomes open to b and b becomes open to a. With the generic channel fairness assumption (see Section 2.3.3) which were used in the verification of the data transfer protocols in Chapter 3, progress of the connection management service cannot be proven. This is so because the opening message can be retransmitted for only a limited period and the the channel is not guaranteed to deliver a message in any finite time. Making stronger assumptions, such that at least one out of k messages are delivered, allows to prove progress. This is the approach used in [LLSA93] to prove the progress of SCMP formally, and in [SL95] to prove the progress of SC informally.

Some protocols such as SNR in Section 3.4 and SC and T/TCP in Section 4.3 have not been verified formally. Operational reasoning based on several case studies and informal arguments was used to establish properties of these protocols. These informal proofs provide lots of information about the behavior of a protocol, but they do not fully substitute a formal verification. The case studies of the informal verifications can greatly simplify a later formal verification using the assertional framework introduced in Chapter 2.

Even the formal verifications of PAWS in Section 3.3 and SCMP in Section 4.2 could be formalized further. With a little more effort, all aspects of the specification language could be formalized. This would have a number of advantages: (i) the specifications could be automatically syntax checked, (ii) the formal specifications could be the basis of conformance testing protocol implementations, and (iii) algorithmic verification support could be used. Rewriting the proofs in [Olá95, OHdG95c] into a machine readable format is likely to be a much more difficult task. The reason for carrying out this task could be the promise of applying automated proof checkers to make sure that the verifications contain no logical errors.

Formal verification provides information about the worst-case behavior of a protocol. The real-time conditions for the correctness of a protocol are often sufficient to decide if the protocol is suitable for a particular purpose or to rank different protocols. In many situations, however, the quantitative behavior is equally or even more important. It is performance analysis that can provide this information. For example, to assess the memory usage of the different cache management policies in Section 4.4.1, performance analysis techniques must be used. Usually both types of analyses must be carried out to make a well-founded decision.

Bibliography

[ACC+94] Paul D. Amer, Christophe Chassot, Thomas J. Connoly, Michael Diaz, and Phillip Conrad. Partial-order transport service for multimedia and other applications. *IEEE/ACM Transactions on Networking*, 2(5):440–455, October 1994.

[All95] Anthony Alles. *ATM Internetworking*. Cisco Systems, May 1995.

[BCS94] Bob Braden, David Clark, and Scott Shenker. Integrated services in the internet architecture: An overview. Request For Comments RFC-1633, USC ISI, MIT LCS, Xerox PARC, June 1994.

[Bel76] Dag Belsnes. Single-message communication. *IEEE Transactions on Communications*, COM-24(2):190–194, February 1976.

[BF93] Ernst W. Biersack and David C. Feldmeier. A timer-based connection management protocol with synchronized clocks and its verification. *Computer Networks and ISDN Systems*, 25(12):1303–1319, July 1993.

[BKKM96] Jacob Brunekef, Joost-Pieter Katoen, Ron Koymans, and Sjouke Mauw. Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Computing*, 9:157–171, 1996.

[Bla83] Richard E. Blahut. *Theory and Practice of Error Control Codes*. Addison-Wesley, Reading, MA, USA, 1983.

[BLCL+94] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk, and Arthur Secret. The World-Wide Web. *Communications of the ACM*, 37(8):76–82, August 1994.

[Bra92] R. Braden. Extending TCP for transactions—concepts. RFC-1379, November 1992.

[Bra93] R. Braden. TCP extensions for high performance: An update. Internet Draft, April 1993. Work in progress.

[Bra94] R. Braden. T/TCP: TCP extensions for transactions, functional specification. RFC 1644, ISI, July 1994.

[Bro93] Manfred Broy. Functional specification of time-sensitive communicating systems. *ACM Transactions on Software Engineering*, 2(1):1–46, 1993.

[Che88] David R. Cheriton. VMTP: Versatile message transaction protocol (protocol specification). RFC-1045, February 1988.

[Che89] David R. Cheriton. Sirpent: A high-performance internetworking approach. In *Proceedings of SIGCOMM'89*, pages 158–169, Austin, TX, September 1989.

[CK74] Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, COM-22(5):637–648, May 1974.

[Cla88] David D. Clark. The design philosophy of the DARPA Internet protocols. In *Proceedings of SIGCOMM'88*, pages 106–114, Stanford, CA, USA, August 1988.

[CM84] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of SIGCOMM'90*, pages 200–208, Philadelphia, PA, September 1990.

[Dal75] Yogen K. Dalal. More on selecting sequence numbers. In *ACM SIGCOMM/SIGOPS Interprocess Communications Workshop*, pages 25–36, Santa Monica, CA, March 1975. Also in *ACM Operating Systems Review*, 9(3), July 1975.

[DDK+90] Willibald A. Doeringer, Doug Dykeman, Matthias Kaiserswerth, Bernd Werner Meister, Harry Rudin, and Robin Williamson. A survey of lightweight transport protocols for high-speed networks. *IEEE Transactions on Communications*, 38(11):2025–2039, November 1990.

[DH95] Stephen E. Deering and Robert M. Hinden. Internet protocol, version 6 (IPv6) specification. Request for Comments 1883, Network Working Group, December 1995.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[Dio94] Christophe Diot. Reliability in multicast services and protocols: A survey. In *International Conference on Local and Metropolitan Communication Systems*, December 1994.

[DJNS93] Bharat T. Doshi, Pravin K. Johri, Arun N. Netravali, and Krishan K. Sabnani. Error and flow control performance of a high speed protocol. *IEEE Transactions on Communications*, 41(5):707–719, May 1993.

[DP93] Martin De Prycker. *Asynchronous Transfer Mode: Solution for Broadband ISDN*. Ellis Horwood, New York, NY, USA, 2nd edition, 1993.

[FB90] David C. Feldmeier and Ernst W. Biersack. Comparison of error control protocols for high bandwidth-delay product networks. In M. Johnson, editor, *Proceedings of the IFIP Workshop on Protocols for High Speed Networks*, pages 271–295, 1990.

[FW78] John G. Fletcher and Richard W. Watson. Mechanisms for a reliable timer-based protocol. *Computer Networks*, 2:271–290, 1978.

[GNS95] Mohamed G. Gouda, Arun N. Netravali, and Krishnan Sabnani. A periodic state exchange protocol and its verification. *IEEE Transactions on Communications*, 43(9):2475–2484, September 1995.

[Gou93] Mohamed G. Gouda. Protocol verification made simple. *Computer Networks and ISDN Systems*, 25:969–980, 1993.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, New York, Heidelberg, Berlin, 1981.

[HHS94] Rainer Haendel, Manfred N. Huber, and Stefan Schroeder. *ATM Networks: Concepts, Protocols, Applications*. Addison-Wesley, Reading, MA, USA, 1994.

[Hui95] Christian Huitema. *Routing in the Internet*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1995.

[Jac88] Van Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM'88*, pages 314–329, Stanford, CA, August 1988.

[Jai92] Raj Jain. Myths about congestion management in high-speed networks. *Journal of Internetworking: Research and Experience*, 3(3):101–113, September 1992.

[Jai95] Raj Jain. Congestion control and traffic management in ATM networks: Recent advances and a survey. Submitted to Computer Networks and ISDN Systems, October 1995.

[JBB92] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC-1323, May 1992.

[Kle92] Leonard Kleinrock. The latency/bandwidth tradeoff in gigabit networks. *IEEE Communications Magazine*, pages 36–40, April 1992.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam94a] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam94b] Leslie Lamport. Verification and specification of concurrent programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pages 347–374, 1994.

[LLSA93] Butler W. Lampson, Nancy A. Lynch, and Jorgen F. Sogaard-Andersen. Correctness of at-most-once message delivery protocols. In *FORTE'93*, pages 387–402, 1993.

[LSW91] Barbara Liskov, Liuba Shrira, and John Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems*, 9(2):125–142, May 1991.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1980.

[Mil91] David L. Mills. Internet time synchronization: the Network Time Protocol. *IEEE Transactions on Communications*, COM-39(10):1482–1493, October 1991.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer-Verlag, New York, NY, USA, 1992.

[MS87] Sandra L. Murphy and A. Udaya Shankar. A verified connection management protocol for the transport layer. In *Proceedings of SIGCOMM'87*, pages 110–125, Stowe, VT, August 1987.

[MS95] David E. McDysan and Darren E. Sophn. *ATM: Theory and Applications*. McGraw-Hill, 1995.

[NRS90] Arun N. Netravali, W. D. Roome, and K. Sabnani. Design and implementation of a high-speed transport protocol. *IEEE Transactions on Communications*, 38(11):2010–2024, November 1990.

[OHdG95a] András L. Oláh and Sonia M. Heemstra de Groot. Analysis of a reliable data-transfer protocol for broadband networks. In *Proceedings of the Australian Telecommunication Networks & Applications Conference, ATNAC'95*, pages 671–676, Sydney, Australia, December 1995.

[OHdG95b] András L. Oláh and Sonia M. Heemstra de Groot. Assertional verification of a connection management protocol. In *Proceedings of the 8th Int'l Conference on Formal Description Techniques*, pages 399–414, Montreal, Canada, October 1995.

[OHdG95c] András L. Oláh and Sonia M. Heemstra de Groot. Assertional verification of a connection management protocol. CTIT Technical Report 95-28, Centre for Telematics and Information Technology, University of Twente, September 1995.

[OHdG96a] András L. Oláh and Sonia M. Heemstra de Groot. Alternative specification and verification of a periodic state exchange protocol. CTIT TR 96-02, Centre for Telematics and Information Technology, January 1996. To appear in *IEEE/ACM Transactions on Networking*.

[OHdG96b] András L. Oláh and Sonia M. Heemstra de Groot. Comments on "Minimum-latency transport protocols with modulo-N incarnation numbers". *IEEE/ACM Transactions on Networking*, 4(4), August 1996.

[OHdG97] András L. Oláh and Sonia M. Heemstra de Groot. Alternative specification and verification of a periodic state exchange protocol. *IEEE/ACM Transactions on Networking*, 5(4), August 1997.

[Olá95] András L. Oláh. Verification of a timestamp-based sliding-window protocol. CTIT Technical Report 95-15, Centre for Telematics and Information Technology, October 1995. An updated version of MI 94-42.

[OP91] S. O'Malley and L. Peterson. TCP extensions considered harmful. RFC-1263, October 1991.

[Pos81a] Jon Postel, editor. Internet protocol. RFC-791, September 1981.

[Pos81b] Jon Postel, editor. Transmission control protocol. RFC-793, September 1981.

[Pro92] Protocol Engines, Inc. *XTP Protocol Definition, Revision 3.6*, January 1992.

[SAGJ93] D. Sanghi, A. K. Agrawala, O. Gudmundsson, and B. N. Jain. Experimental assessment of end-to-end behavior on Internet. In *Proceedings of IEEE Infocom'93*, pages 867–874, San Francisco, CA, USA, March 1993.

[SD78] Carl A. Sunshine and Yogen K. Dalal. Connection management in transport protocols. *Computer Networks*, 2:454–473, 1978.

[Sha89] A. Udaya Shankar. Verified data transfer protocols with variable flow control. *ACM Transactions on Computer Systems*, 7(3):281–316, August 1989.

[Sha91] A. Udaya Shankar. Modular design principles for protocols with an application to the transport layer. *Proceedings of the IEEE*, 79(12):1687–1707, December 1991.

[Sha93] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.

[Sha94] A. Udaya Shankar. Reasoning assertionally about real-time systems. *Proceedings of the IEEE*, 82(1):172–183, January 1994.

[SL93] A. Udaya Shankar and David Lee. Modulo-N incarnation numbers for cache-based transport protocols. In *International Conference on Network Protocols*, pages 46–54, San Francisco, CA, October 1993.

[SL95] A. Udaya Shankar and David Lee. Minimum-latency transport protocols with modulo-N incarnation numbers. *IEEE/ACM Transactions on Networking*, 3(3):255–268, June 1995.

[Sta90] William Stallings. *Handbook of Computer Communications Standards, Volume 1: The Open Systems Interconnection (OSI) Model and OSI-Related Standards*. Macmillan, New York, NY, USA, 2nd edition, 1990.

[Sta95] William Stallings. *ISDN and Broadband ISDN with Frame Relay and ATM*. Prentice-Hall, Englewood Cliffs, NJ, USA, 3rd edition, 1995.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, MA, USA, 1994.

[Ste95] Martha Steenstrup, editor. *Routing in Communications Networks*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1995.

[Ste96] W. Richard Stevens. *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*. Addison-Wesley, Reading, MA, USA, 1996.

[Tan89] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall International, Inc., Englewood Cliffs, NJ, 2nd edition, 1989.

[Tel91] Gerard Tel. *Topics in Distributed Algorithms*. Cambridge University Press, Cambridge, UK, 1991.

[Tom75] Raymond S. Tomlinson. Selecting sequence numbers. In *ACM SIGCOMM/SIGOPS Interprocess Communications Workshop*, pages 11–23, Santa Monica, CA, March 1975. Also in *ACM Operating Systems Review*, 9(3), July 1975.

[Wat81] Richard W. Watson. Timer-based mechanisms in reliable transport protocol connection management. *Computer Networks*, 5:47–56, 1981.

[Wat89] Richard W. Watson. The Delta-t transport protocol: Features and experience. In H. Rudin and R. Williamson, editors, *Protocols for High-Speed Networks*, pages 3–17, 1989.

[WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, Reading, MA, USA, 1995.

[XTP95] XTP Forum, Santa Barbara, CA, USA. *Xpress Transport Protocol Specification, XTP Revision 4.0*, March 1995.

[Zha86] Lixia Zhang. Why TCP timers don't work well. In *Proceedings of SIGCOMM'86*, Stowe, VT, USA, August 1986.

Appendix A

Proofs of Section 3.3

In this appendix the details of verifying PAWS are presented. The overall structure of verifying PAWS was discussed in Section 3.3. The proofs presented here are taken from the report [Olá95] with minor modifications.

Since many of the assertions are in the form of $\Box P$ and $P \rightarrow Q$, we will have to use the invariance rule (2.1) quite often. The use of this rule requires us to show that $(\forall e \in E : \{P\}e\{P\})$ holds.

To make it easy to handle these proofs, we will use X and X' to denote the value of state variable X immediately before and after the occurrence of an event. Similarly, P' denotes the assertion P with all its state variables replaced by their primed versions. We say that the assertion P is not affected by event e if none of the state variables in P are updated by $action(e)$.

Note that the channel events do not invalidate any of the invariants. The only reference to the contents of a channel C is in the form of $\Box(P \in C \Rightarrow Q)$, where P is a packet and Q is predicate that may depend on the value of P. The channel events cannot violate such assertions because none of them produces a packet which is different from the packets already in the channel.

A.1 Safety

A.1.1 Correct interpretation conditions

The assertions (A.1)–(A.5) reformulate the CI requirements (3.41)–(3.46) from Section 3.3.4 in a form that is easier to handle. Their usefulness is expressed by Lemma A.1 below.

$$\Box(D \in C_{S,R} \wedge D.Ts \geq R.TsRec \Rightarrow$$
$$\Rightarrow \quad R.Nxt + RW \geq D.Seq + D.Len \geq D.Seq \geq R.Nxt + RW - N_S + 1) \quad \text{(A.1)}$$

$$\Box(A \in C_{R,S} \wedge A.TsEcho \geq S.TsEchoRec \Rightarrow$$
$$\Rightarrow \quad S.Nxt \geq A.Ack \geq S.Nxt - N_S + 1) \quad \text{(A.2)}$$

$$\Box(D \in C_{S,R} \wedge R.RecOld \Rightarrow D.Ts \geq R.TsRec) \quad \text{(A.3)}$$

$$\Box(S.EchoRecOld \Rightarrow S.Clk - TW_S \geq S.TsEchoRec) \quad \text{(A.4)}$$

$$\Box(A \in C_{R,S} \Rightarrow S.Clk \geq A.TsEcho \geq S.Clk - TW_S + 1) \quad \text{(A.5)}$$

Lemma A.1

(a) $(A.1) \wedge (A.3) \Rightarrow (3.41)$*;*
(b) $(A.2) \wedge (A.4) \wedge (A.5) \Rightarrow (3.42)$*;*
(c) $(A.5) \Rightarrow (3.44)$.

Proof: $R.RecOld \vee D.Ts \geq R.TsRec$ is in the antecedent of (3.41). In case of $R.RecOld =$ `true`, (A.3) implies the antecedent of (A.1) which implies (3.41). In case $\neq R.RecOld$, (3.41) and (A.1) are equivalent.

Similarly, if $S.EchoRecOld =$ `true`, then $A.TsEcho \geq S.TsEchoRec$ is implied by (A.4) and (A.5). Because of (A.2), (3.42) is satisfied in this case. If $S.EchoRecOld =$ `false`, then (3.42) and (A.2) are equivalent.

$(A.5) \Rightarrow (3.44)$ is true because the parameter TW_S is represented as a modulo-N_C number, thus $TW_S < N_C$ must hold. □

A.1.2 Real-time assumptions

The assertions below formulate some straightforward properties of the epoch variables $S.t_S$, $S.t_C$, and $R.t_C$.

$$\Box(S.t_S[0 \ldots S.Nxt - 1] \neq \lambda \wedge S.t_S[S.Nxt \ldots \infty] = \lambda) \quad \text{(A.6)}$$

$$\Box(S.t_C[0 \ldots S.Clk - 1] \neq \lambda \wedge S.t_C[S.Clk \ldots \infty] = \lambda) \quad \text{(A.7)}$$

$$\Box(R.t_C[0 \ldots R.Clk-1] \neq \lambda \wedge R.t_C[R.Clk \ldots \infty] = \lambda) \tag{A.8}$$

$$\Box(n \in [1 \ldots S.Clk-1] \Rightarrow \gamma_S < S.t_C[n] - S.t_C[n-1] < \Gamma_S) \tag{A.9}$$

$$\Box(n \in [1 \ldots R.Clk-1] \Rightarrow \gamma_R < R.t_C[n] - R.t_C[n-1] < \Gamma_R) \tag{A.10}$$

Lemma A.2 *(A.6)–(A.10) are invariants of the protocol provided that (3.47)–(3.50) are satisfied.*

Proof: The assertions (A.6)–(A.10) hold in the initial state of the system which can be checked by substitution.

S.ClockTick affects (A.7) and (A.9). (A.7)' holds because $S'.Clk = S.Clk + 1$, only $S.t_C[S.Clk]$ is changed to a non-λ value and (A.7) was true before the event. The lower limit on $S.t_C[n] - S.t_C[n-1]$ is simply the implication of (3.47) and (A.6) because for any $n \in [0 \ldots S.Clk-1]$, $S.t_C[n] \neq \lambda$. Because of (A.8), we know that in case of $n \in [1 \ldots S'.Clk-2]$ the upper limit also holds. $S'.t_C[S'.Clk-1] - S'.t_C[S'.Clk-2] < \Gamma_S$ is implied by (3.48), $S.t_C[S'.Clk-1] = \tau$ and the increasing time axiom (2.9).

We can prove in the very same way that *R.ClockTick* preserves (A.8) and (A.10). Proving that *Accept*(*data*) preserves (A.6) is similar to proving that *S.ClockTick* preserves (A.7). Other protocol events do not affect any of the invariants (A.6)–(A.10). □

A.1.3 Correct interpretation of sequence numbers

The assertions below complement the assertions (3.51)–(3.59). These two sets of assertions can then be used to prove the invariance of the CI conditions for the sequence numbers as it is stated by Lemma A.4.

$$\Box(Source[n] = \texttt{empty} \Leftrightarrow S.TsMin[n] = \texttt{empty} \Leftrightarrow S.TsMax[n] = \texttt{empty}) \tag{A.11}$$

$$\Box(n \in [0 \ldots S.Nxt-1] \Rightarrow S.t_C[S.TsMin[n]-1] \leq S.t_S[n]) \tag{A.12}$$

$$\begin{aligned}\Box(&n \in [0 \ldots S.Nxt-1] \wedge S.Clk > S.TsMin[n] \Rightarrow \\ &\Rightarrow \quad S.t_S[n] \leq S.t_C[S.TsMin[n]])\end{aligned} \tag{A.13}$$

$$\Box(CA \in C_{R,S} \Rightarrow CA.Ack \leq R.LAck) \tag{A.14}$$

$$\Box(S.Una \leq R.LAck \leq R.Nxt \wedge R.Nxt - RW \leq R.LAck) \tag{A.15}$$

$$\begin{aligned}\Box(&D \in C_{S,R} \Rightarrow D.Seq + D.Len - 1 \leq D.SnMax \leq D.Seq + RW - 1 \wedge \\ &\wedge \quad D.SnMax \leq S.Nxt - 1)\end{aligned} \tag{A.16}$$

$$\Box(D_{1,2} \in C_{S,R} \wedge D_1.Ts > D_2.Ts \Rightarrow D_1.SnMax \geq D_2.SnMax) \tag{A.17}$$

$$\Box(D \in C_{S,R} \wedge D.Ts > R.TsRec \Rightarrow D.SnMax \geq R.SnMax) \tag{A.18}$$

$$\Box(R.Nxt > R.LAck \Rightarrow R.SnMax \geq R.LAck) \tag{A.19}$$

$$\Box(R.Nxt = R.LAck \Rightarrow R.SnMax \geq R.LAck - RW) \tag{A.20}$$

$$\begin{aligned}\Box(&CA \in C_{R,S} \Rightarrow CA.Ack - RW \leq CA.SnMax \leq \\ &\leq \quad \min\{CA.Ack + CA.Wnd - 1, S.Nxt - 1\})\end{aligned} \tag{A.21}$$

$$\Box(SA \in C_{R,S} \Rightarrow SA.SnMax \leq \min\{SA.Ack + RW - 2, S.Nxt - 1\}) \tag{A.22}$$

Lemma A.3 *(3.51)–(3.59) and (A.11)–(A.22) are invariants of the protocol.*

Proof: By simple substitution into the proof rules (see [Olá95]). □

The following lemma states our first constraint on the real-time parameters of the protocol:

Lemma A.4 *The inequality (3.60) is a sufficient condition for the correct interpretation of sequence numbers, i.e.,*

$$(3.60) \wedge (3.6)\text{–}(3.12) \wedge (3.51)\text{–}(3.59) \wedge (A.11)\text{–}(A.22) \Rightarrow (A.1) \wedge (A.2)$$

Proof: (3.7) and (A.16) imply that $D.Seq + D.Len \leq S.Nxt$ and using (3.6) we get $D.Seq \leq D.Seq + D.Len \leq R.Nxt + RW$ which is the first part of (A.1).

Now we have to prove the lower limit on $D.Seq$. Applying (3.54) we get:

$$D.Ts \leq S.TsMax[D.SnMax]$$

From (3.55):

$$S.TsMin[R.SnMax] \leq R.TsRec$$

Combining these two, we get:

$$D.Ts \geq R.TsRec \Rightarrow S.TsMax[D.SnMax] \geq S.TsMin[R.SnMax]$$

and applying (3.52) and (3.53) for the above:

$$S.TsMax[D.SnMax] \geq S.TsMin[R.SnMax] \Rightarrow D.SnMax \geq R.SnMax - \lceil \Gamma_S \cdot B \rceil$$

Using (A.16), i.e., $D.SnMax \leq D.Seq + RW - 1$, and (3.58), i.e., $R.SnMax \geq R.Nxt - \lceil \Gamma_S \cdot B \rceil$, we get:

$$D.Ts \geq R.TsRec \Rightarrow D.Seq \geq R.Nxt - \lceil \Gamma_S \cdot B \rceil - 2RW + 1$$

Therefore we can see that $N_S \geq 3RW + \lceil \Gamma_S \cdot B \rceil$ is sufficient to satisfy (A.1) which yields (3.60).

(A.2) can be proven by using the same steps for the reverse channel. (3.1), (3.8), and (3.9) imply the upper limit, $A.Ack \leq S.Nxt$. For the lower limit, from (3.53), (3.56), and (3.57) we get

$$A.TsEcho \geq S.TsEchoRec \Rightarrow A.SnMax \geq S.SnMax - \lceil \Gamma_S \cdot B \rceil$$

and using (3.6), (3.8), and (3.59)

$$A.TsEcho \geq S.TsEchoRec \Rightarrow A.Ack \geq S.Nxt - \lceil \Gamma_S \cdot B \rceil - 3RW + 1$$

Therefore, $N_S \geq 3RW + \lceil \Gamma_S \cdot B \rceil$ is sufficient for (A.2) to hold, as well. □

A.1.4 Correct interpretation of timestamps

The following assertions are needed to prove the invariance of the CI conditions for timestamps:

$$\Box(S.TsEchoRec \leq R.TsRec \leq S.Clk) \tag{A.23}$$

$$\Box(D \in C_{S,R} \Rightarrow D.Ts \leq S.Clk) \tag{A.24}$$

$$\Box(A \in C_{R,S} \Rightarrow A.TsEcho \leq R.TsRec \wedge A.EchoExp \leq R.TsExp) \tag{A.25}$$

$$\Box(R.TsExp \leq R.Clk) \tag{A.26}$$

$$\Box(\langle D, t\rangle \in C_{S,R} \Rightarrow S.t_C[D.Ts - 1] \leq t) \tag{A.27}$$

$$\Box(\langle D, t\rangle \in C_{S,R} \wedge S.t_C[D.Ts] \neq \lambda \Rightarrow t \leq S.t_C[D.Ts]) \tag{A.28}$$

$$\Box(R.t_C[R.TsExp] \neq \lambda \Rightarrow S.t_C[R.TsRec - 1] \leq R.t_C[R.TsExp]) \tag{A.29}$$

$$\Box(S.t_C[R.TsRec] \neq \lambda \Rightarrow R.t_C[R.TsExp - 1] \leq S.t_C[R.TsRec] + L_{S,R}) \tag{A.30}$$

$$\begin{aligned}&\Box(\langle A, t\rangle \in C_{R,S} \wedge R.t_C[A.EchoExp + TW_R - 1] \neq \lambda \Rightarrow \\ &\quad\Rightarrow \quad t \leq R.t_C[A.EchoExp + TW_R - 1])\end{aligned} \tag{A.31}$$

$$\begin{aligned}&\Box(\langle A, t\rangle \in C_{R,S} \wedge R.t_C[A.EchoExp + TW_R - 1] \neq \lambda \Rightarrow \\ &\quad\Rightarrow \quad t \leq R.t_C[A.EchoExp + TW_R - 1])\end{aligned} \tag{A.32}$$

$$\Box(\langle A, t\rangle \in C_{R,S} \Rightarrow S.t_C[A.TsEcho - 1] \leq t \wedge R.t_C[A.EchoExp - 1] \leq t) \tag{A.33}$$

$$\begin{aligned}&\Box(A \in C_{R,S} \wedge S.t_C(A.TsEcho) \neq \lambda \Rightarrow \\ &\quad\Rightarrow \quad R.t_C[A.EchoExp - 1] \leq S.t_C[A.TsEcho] + L_{S,R})\end{aligned} \tag{A.34}$$

Lemma A.5 *(3.45) and (A.23)–(A.34) are invariants of the protocol.*

Note that (3.45) is included in the lemma because (3.45) is a precondition of (A.31) and (A.32) with respect to the *SendACK* and *SendSACK* events. The invariance of (3.45) and (A.23)–(A.34) can be proven by simple application of the proof rules and without using the real-time assumptions (see [Olá95]).

The inequalities below are similar to the inequalities (3.61)–(3.63) with the only difference that here we make a distinction between the delays in the two channels. It is easy

to see that (3.61)–(3.63) can be obtained from (A.35)–(A.37) by simply substituting $L = L_{S,R} = L_{R,S}$.

$$N_C \geq \left\lceil \frac{L_{S,R} + TW_R \cdot \Gamma_R}{\gamma_S} \right\rceil + \left\lceil \frac{L_{S,R}}{\gamma_S} \right\rceil + 3 \tag{A.35}$$

$$TW_R \geq \left\lceil \frac{L_{S,R}}{\gamma_R} \right\rceil + 1 \tag{A.36}$$

$$TW_S \geq \left\lceil \frac{L_{S,R} + TW_R \cdot \Gamma_R + L_{R,S}}{\gamma_S} \right\rceil + 1 \tag{A.37}$$

The following lemma states that (A.35)–(A.37) are sufficient for the correct interpretation of timestamps. Note that (a) and (c) state that the appropriate CI condition is implied by invariants and some extra restrictions. On the other hand, (b) states somewhat less: (A.36) implies that *R.ClockTick* does not invalidate (A.3), but we still have to show that other events also preserve it.

Lemma A.6 *The inequalities (A.35)–(A.37) are sufficient conditions for the invariance of (3.43), (A.31), and (A.33):*

(a) $(A.35) \wedge (A.23)$–$(A.34) \Rightarrow (3.43)$*;*

(b) $(A.36) \wedge (A.23)$–$(A.34) \Rightarrow \{(A.3)\}R.ClockTick\{(A.3)\}$*;*

(c) $(A.37) \wedge (A.23)$–$(A.34) \Rightarrow (A.5)$.

We prove Lemma A.6 is three steps:

Case (a)

Proof: Let us start with the first part of (3.43) which requires that there is an upper limit of $D.Ts$ if the receiver's recent timestamp is still valid. From the increasing time axiom (2.9), (A.27) and $\langle D, t\rangle \in C_{S,R}$ we get

$$S.t_C[D.Ts - 1] \leq t \leq \tau$$

Using the fact that $R.RecOld =$ `false` and (3.45):

$$R.Clk \leq R.TsExp + TW_R - 1$$

and substituting this into (A.8) results in

$$R.t_C[R.TsExp + TW_R - 1] = \lambda$$

Combining the above inequalities with (3.50) and (A.10) leads to

$$\tau \leq R.t_C[R.TsExp - 1] + TW_R \cdot \Gamma_R$$

Using the first inequality, we get

$$S.t_C[D.Ts - 1] \leq R.t_C[R.TsExp - 1] + TW_R \cdot \Gamma_R$$

Let us first assume that $S.t_C[R.TsRec] = \lambda$. In this case,

$$(A.6) \Rightarrow S.Clk \leq R.TsRec \wedge (A.23) \Rightarrow S.Clk \geq R.TsRec$$

thus $R.TsRec = S.Clk$ must hold. Since $D.Ts \leq S.Clk$ due to (A.24), in this case (3.43) is satisfied.

The other case is when $S.t_C[R.TsRec] \neq \lambda$. Now we can use (A.30) from which we get

$$R.t_C[R.TsExp - 1] \leq S.t_C[R.TsRec] + L_{S,R}$$

Combining this with earlier results, we get

$$S.t_C[D.Ts - 1] - S.t_C[R.TsRec] \leq L_{S,R} + TW_R \cdot \Gamma_R$$

Because of the limit on the minimum clock tick length (3.47), the above inequality implies

$$D.Ts \leq R.TsRec + \left\lceil \frac{L_{S,R} + TW_R \cdot \Gamma_R}{\gamma_S} \right\rceil + 1$$

therefore, the upper limit of $D.Ts$ in (3.43) is satisfied if

$$K_R \geq \left\lceil \frac{L_{S,R} + TW_R \cdot \Gamma_R}{\gamma_S} \right\rceil + 1 \tag{A.38}$$

The second step is to establish the sufficient conditions for the lower limit of $D.Ts$ in (3.43). Let us assume that $S.t_C[D.Ts] = \lambda$. In this case, we know that $D.Ts = S.Clk$ must hold, thus $D.Ts \geq R.TsRec$ also holds.

Let us now examine the case when $S.t_C[D.Ts] \neq \lambda$. From (A.28) and (2.13) we get

$$S.t_C[R.TsRec - 1] \leq \tau \leq S.t_C[D.Ts] + L_{S,R}$$

Following the same argument as in the first step of the proof, we get

$$R.TsRec - \left\lceil \frac{L_{S,R}}{\gamma_S} \right\rceil - 1 \leq D.Ts$$

and thus the required lower limit on $D.Ts$ is implied by

$$K_R \leq N_C - \left\lceil \frac{L_{S,R}}{\gamma_S} \right\rceil - 2 \tag{A.39}$$

Since the specific value of K_R is not important, the existence of K_R is sufficient, we can combine (A.38) and (A.39) into a single inequality which is exactly (A.35). □

Case (b)

Proof:

We want to prove that *R.ClockTick* preserves (A.3) if (A.36) holds. (A.3)' holds if $R'.RecOld = \texttt{false}$. If $R'.RecOld = R.RecOld = \texttt{true}$, then (A.3)' is implied by (A.3). The only remaining case is when $R'.RecOld = \texttt{true}$ and $R.RecOld = \texttt{false}$. In this case, $R'.Clk = R.Clk + 1 = R.TsRec + TW_R$. (3.49)' must hold after the event, because it is an assumption. From this we get

$$R'.t_C[R'.Clk - 1] - R.t_C[R.TsExp] > (TW_R - 1) \cdot \gamma_R$$

Because $R.TsExp < R.Clk$, we get

$$S.t_C[R.TsRec - 1] \leq R.t_C[R.TsExp]$$

by applying (A.29). Altogether, we have

$$R'.t_C[R.Clk] = \tau > S.t_C[R.TsRec - 1] + (TW_R - 1) \cdot \gamma_R$$

Let us now suppose, that there is a packet $\langle D, t \rangle$ in $C_{S,R}$ for which $D.Ts < R.TsRec$. Applying (A.28) for this packet, we get

$$t \leq S.t_C[D.Ts] \leq S.t_C[R.TsRec - 1]$$

Combining this with the previous result yields

$$\tau - t > (TW_R - 1) \cdot \gamma_R$$

Because (2.13) states that no packet can stay longer than $L_{S,R}$ in $C_{S,R}$,

$$(TW_R - 1) \cdot \gamma_R \geq L_{S,R}$$

is sufficient to assure that no such packet like D can exist in the channel. This condition is equivalent to (A.36). □

Case (c)

Proof: The upper limit on $A.TsEcho$ is implied by (A.23) and (A.25). To prove the lower limit, let us consider a packet $\langle A, t\rangle \in C_{R,S}$. If $S.t_C[A.TsEcho] = \lambda$, then $A.TsEcho \geq S.Clk$ must hold because of (A.7). On the other hand, (A.23) and (A.25) imply that $A.TsEcho \leq S.Clk$, thus $A.TsEcho = S.Clk$ in this case. Let us consider now when $S.t_C[A.TsEcho] \neq \lambda$. In this case, we get

$$R.t_C[A.EchoExp - 1] \leq S.t_C[A.TsEcho] + L_{S,R}$$

from (A.34). Applying (A.32) for the above inequality yields

$$t \leq S.t_C[A.TsEcho] + L_{S,R} + TW_R \cdot \Gamma_R$$

In order to satisfy (A.5), we want to assure that $A.TsEcho > S.Clk - TW_S$. (2.13) says that $\tau - t < L_{R,S}$; from (A.7) and the increasing time axiom, we know that $\tau \geq S.t_C[S.Clk - 1]$. Therefore,

$$S.t_C[S.Clk - 1] \geq S.t_C[S.Clk - TW_S] + L_{S,R} + TW_R \cdot \Gamma_R + L_{R,S}$$

is sufficient to assure that no acks with $A.TsEcho \leq S.Clk - TW_S$ exist. Because of (A.7), (A.37) is sufficient for the above inequality to hold. □

Now we have enough results that we can prove the following theorem which states that the protocol satisfies its safety requirements:

Theorem A.7 *The correct interpretation conditions (3.41)–(3.46) are invariants of the protocol provided that the real-time requirements (3.60)–(3.63) are satisfied.*

The statement of the theorem follows from the lemmas we have proven so far. This completes the proof of the protocol's safety.

A.2 Progress

The progress properties of the protocol are stated by the following theorem:

Theorem A.8 *The assertions (3.13)–(3.15), (3.30)–(3.40), and (A.40) are satisfied by the system provided that the liveness assumptions (3.27)–(3.29) hold.*

Assertion (A.40) is another assertion that is used in the proof of Theorem A.8:

$$R.Nxt \geq n \rightsquigarrow S.Una \geq n \tag{A.40}$$

Proof: In order to simplify the description of the proofs, we will always assume that assertions are written in the form of $A \rightsquigarrow B$ or $A \rightarrow B$, whenever possible. This means, for example, that $A = S.Nxt > R.Nxt = n$ and $B = R.Nxt > n$ during the proof of (3.13).

For the proof of (3.32), let A and B denote $S.Una = n \wedge S.Clk = m$ and $D \in C_{S,R} \wedge D.Ts > m \Rightarrow D.Seq \geq n$, respectively. $A \Rightarrow B$ holds because of the invariant $\Box(D \in C_{S,R} \Rightarrow D.Ts \leq S.Clk)$ which is implied by (3.51), (3.54), and (A.16). We know that $S.Una$ and $S.Clk$ are monotone increasing, thus it is enough to prove that

$$\{S.Una \geq n \wedge S.Clk \geq m \wedge B\}e\{B\}$$

holds for any event e. Only the *SendD* and *SendP* events could violate this assertion, but $S.Una \geq n$ assures that B holds after the event.

Let us write (3.33) in the form of $A \rightarrow B$. $A \Rightarrow B$ follows from (3.51) and (3.53). We also know that $S.Una$, $S.Nxt$ and $S.Clk$ are monotone increasing, thus it is enough to establish the truth of the following statements for any event e of the system:

$$\{B \wedge S.Nxt > S.Una = n \wedge S.Clk \geq m\}e\{B\}$$

$$\{B \wedge S.Nxt \geq S.Una > n \wedge S.Clk \geq m\}e\{B\}$$

The first statement is true because the precondition implies that *SendP*, which could generate zero-length packets, is disabled. In the second case no packets with $D.Seq = n$ can be generated because $S.Una > n$ is in the precondition.

In case of (3.34), $A \Rightarrow B$ follows from (A.23). $R.Nxt$ and $R.TsRec$ can only be changed in the *RecD* event, thus it is enough if we show that $\{B\}RecD\{B\}$. Because $S.Una$ is monotone increasing, $R.Nxt = n \Rightarrow S.Una = n$. Using (A.15) we also get $R.LAck = n$. B could be violated after the occurrence of a $RecD(D)$ event if $D.Ts > m$ holds. From

(3.32) we know however, that in this case $D.Seq \geq n$. If $D.Seq > n = R.LAck$, then $R.TsRec$ is not updated; if $D.Seq = n$, then $D.Len > 0$ because of (3.33), thus $R.Nxt > n$ will hold which means that B holds vacuously.

(3.30) can now be proven with the help of (3.34). The *leads-to via message set rule* (2.12) can be used to prove (3.30). Let A and B denote $S.Nxt > R.Nxt = S.Una = n$ and $R.Nxt > n$, respectively. We prove that

$$A \wedge S.Clk = m \leadsto B$$

via message set $M = \{D : D.Type = DATA, D.Seq = n, D.Len > 0, D.Ts \geq m\}$. $\{A\}e\{A \vee B\}$ holds for every event because of the monotonicity of $S.Nxt$, $R.Nxt$ and $S.Una$. We can also show that the $RecD(M)$ event will establish B. Because of (3.34) we only have to show that

$$\{A \wedge R.TsRec \leq m\}RecD(M)\{B\}$$

which is easy to verify. Finally, we have to show that

$$A \wedge S.Clk \geq m \wedge count(M) = k \leadsto B \vee count(M) > k)$$

This assertion is implied by the liveness assumption (3.27). By now, we have proven that $A \wedge S.Clk = m \leadsto B$ for any value of m, thus $A \leadsto B$ also holds.

(3.35) is also proved from the *leads-to via message set rule* with $A = R.Nxt > S.Una = n \wedge S.Clk \geq m$, $B = S.Una > n \vee (\neg R.RecOld \wedge R.TsRec \geq m)$ and $M = \{D : D.Seq = n, D.Ts \geq m\}$. $\{A\}e\{A \vee B\}$ is easy to check, $\{A\}RecD(M)\{B\}$ holds because $R.LAck \geq S.Una = n$ and $R.TsRec$ is updated if $D.Seq \leq R.LAck$ holds. (3.27) assures that messages from M will be sent until progress is made, i.e., B is established.

The proof of (3.36) is basically the same as the proof of (3.32); it is based on the monotone increasing property of $R.Nxt$ and $R.TsRec$. $A \Rightarrow B$ holds because of the invariant $\Box(A \in C_{R,S} \Rightarrow A.TsEcho \leq R.TsRec)$ which is part of (A.25). Because $R.Nxt$ and $R.TsRec$ are monotone increasing, it is enough to show that

$$\{B \wedge R.Nxt \geq n \wedge R.TsRec \geq m\}e\{B\}$$

holds for any event e. It is easy to check that both of the *SendACK* and *SendSACK* events satisfy this rule.

The assertion (3.37) and its proof are very similar to the assertion (3.34). $A \Rightarrow B$ is implied by (A.23). Using (3.36) for the case $R.Nxt = l > S.Una = n \wedge R.TsRec = m$, we get $A \in C_{R,S} \wedge A.TsEcho > m \Rightarrow A.Ack \geq l$. Thus it is enough to show that

$$\{B \wedge (A \in \wedge A.TsEcho > m \Rightarrow A.Ack > n)\}e\{B\}$$

holds for any event. Only the *RecACK* event can update $S.TsEchoRec$, but in case $S.TsEchoRec$ is updated then $S.Una$ is also updated to $A.Ack$, thus B will hold.

In order to prove (3.14), we first establish that $(A \rightsquigarrow B) \vee (A \rightarrow A)$. This is so because $\{A\}e\{A \vee B\}$ holds for every event. Then we show that $A \rightarrow A$ leads to contradiction, thus $A \rightsquigarrow B$ must hold. From (3.35) and (3.29) we get

$$\begin{aligned}
&\Box(A \wedge S.Clk \geq m) \Rightarrow \\
&\quad \Rightarrow \quad \Box\Diamond(\neg R.RecOld \wedge R.TsRec \geq m) \\
&\quad \Rightarrow \quad (count(N) = k \rightsquigarrow count(N) > k)
\end{aligned}$$

where $N = \{A : A.Ack > n \wedge A.TsEcho \geq m\}$. The channel-fairness assumption assures that messages from the set N are eventually delivered to the sender S. Using (3.37), it is easy to show that the reception of any of these messages causes B to hold after the occurrence of the *RecACK* event. Thus $A \rightsquigarrow B$ must hold.

(3.38) can be proven in the same way as (3.35), but in this case the liveness assumption (3.28) has to be used.

(3.38) can be proven from the *leads-to via message set rule* (2.12) with $A = S.Una + S.Wnd = S.Una = n \wedge S.Clk \geq m$, $B = S.Una + S.Wnd > n \vee (\neg R.RecOld \wedge R.TsRec \geq m)$ and $M = \{D : D.Seq = n \wedge D.Ts \geq m\}$. $\{A\}e\{A \vee B\}$ can be checked for each event, $\{A\}RecD(M)\{B\}$ holds because $A \Rightarrow R.LAck = n$ and $R.TsRec$ is updated when $D.Seq \leq R.LAck$ holds. (3.28) assures that messages from M are repeatedly sent until B is established.

In case (3.39), $A \Rightarrow B$ holds because of the invariant (A.25). We also know that $R.Nxt + R.Wnd$ and $R.TsRec$ are monotone increasing, thus it is enough to show that

$$\{B \wedge R.Nxt + R.Wnd > n \wedge R.TsRec \geq m\}e\{B\}$$

holds for every event e. The *SendACK* event satisfies this statement because $A.Ack + A.Wnd = R.Nxt + R.Wnd$ holds right after the occurrence of the event.

(3.40) is proven from (3.39) in the same way as (3.37) was proven from (3.36). (A.40) can be proven from the *leads-to well founded closure rule* (2.8) because

(i) $R.Nxt \geq n \rightarrow R.Nxt \geq n$;

(ii) $\Box(S.Una \geq n \vee (\exists x : x > 0 : S.Una = n - x)$;

(iii) $R.Nxt \geq n \wedge S.Una = n - w \rightsquigarrow S.Una \geq n \vee (\exists x : w > x > 0 : S.Una = n - x)$.

(i) is the monotone increasing property of $R.Nxt$, (ii) is trivial, and (iii) is an alternative form of (3.14) which we have proven already.

(3.13) and (3.15) can be proven with the help of (A.40). (3.15) holds because of (3.31) and the following assertion which comes from (A.40):

$$R.Nxt = n \wedge R.Wnd > 0 \rightsquigarrow S.Una \geq n \wedge R.Nxt + R.Wnd > n$$

$S.Una \geq n \Rightarrow S.Una + S.Wnd \geq n$. If $S.Una + S.Wnd = n$, then we can apply (3.31):

$$S.Una + S.Wnd = S.Una = n \wedge R.Nxt + R.Wnd > n \rightsquigarrow S.Una + S.Wnd > n$$

Similarly, (3.13) holds because

$$S.Nxt > R.Nxt = n \rightsquigarrow S.Nxt > n \wedge S.Una \geq n$$

If $R.Nxt > n$, then we have proven (3.13), if $R.Nxt = n$, then $S.Una = n$ must hold as well, and (3.30) implies (3.13). This completes the proof of Theorem A.8. □

Appendix B

Proofs of Section 4.2

B.1 Proof of Lemma 4.1

Here we want to show that the safety requirements of SCMP (4.2)–(4.4) imply the generic safety requirement (4.1) for any CM service. Just for the reference we present the assertions in question once more:

$$\Box((\mathit{connected}_{[a,b]}(i,j) \wedge \mathit{connected}_{[a,b]}(k,l)) \Rightarrow (i = k \Leftrightarrow j = l)) \tag{B.1}$$

$$S.CR[b,a].\mathit{open\_to}[i] = j \rightarrow C.CR[a,b].\mathit{open\_to}[j] \in \{i, \texttt{empty}\} \tag{B.2}$$

$$C.CR[a,b].\mathit{open\_to}[i] = j \rightarrow S.CR[b,a].\mathit{open\_to}[j] = i \tag{B.3}$$

$$S.CR[b,a].\mathit{open\_to}[i] = j \wedge k \neq i \rightarrow S.CR[b,a].\mathit{open\_to}[k] \neq j \tag{B.4}$$

All the above assertions are written in such a way that the hosts of endpoint a and b are denoted by C and S, respectively.

All we have to show is that whenever the antecedent of (B.1) holds, then its consequent is implied by the assertions (B.2), (B.3) and (B.4). Using the definition of $\mathit{connected}_{[a,b]}$, we expand the antecedent of the implication in (B.1):

$$\begin{aligned}&(C.CR[a,b].\mathit{open\_to}[i] = j \vee S.CR[b,a].\mathit{open\_to}[j] = i) \wedge \\ &\quad \wedge \quad (C.CR[a,b].\mathit{open\_to}[k] = l \vee S.CR[b,a].\mathit{open\_to}[l] = k)\end{aligned} \tag{B.5}$$

Based on (B.5), we have to consider four cases:

1. $C.CR[a,b].open\_to[i] = j \wedge C.CR[a,b].open\_to[k] = l$

   It is trivial that $i = k \Rightarrow j = l$ in this case. In case of $j = l$, we can use (B.3). The assumptions of this case plus (B.3) imply that $i = S.CR[b,a].open\_to[j] = S.CR[b,a].open\_to[l] = k$ which concludes this case.

2. $C.CR[a,b].open\_to[i] = j \wedge S.CR[b,a].open\_to[l] = k$

   $S.CR[b,a].open\_to[l] = k$ and (B.2) imply that $C.CR[a,b].open\_to[k] = l$ or `empty` If $i = k$, then $C.CR[a,b].open\_to[i] = j \neq$ `empty` implies that $j = l$. If $j = l$ then $C.CR[a,b].open\_to[i] = j = l$ and (B.3) implies that $S.CR[b,a].open\_to[l] = i$, thus $i = k$.

3. $S.CR[b,a].open\_to[j] = i \wedge C.CR[a,b].open\_to[k] = l$

   This is identical to the previous case if the indices are permuted.

4. $S.CR[b,a].open\_to[j] = i \wedge S.CR[b,a].open\_to[l] = k$

   If $i = k$, then (B.4) would be violated if j were not equal to l. If $j = l$, then $i = k$ trivially.

B.2 Proof of Theorem 4.2

The statement we have to prove is that the assertions (4.2)–(4.13) are satisfied by SCMP. We will use the proof rules of Chapter 2 to transform the assertions into simpler forms until we obtain a number of Hoare-triples in the form $\{F\}e\{F\}$ which must hold for every event e in the system.

There are six assertions, namely (4.2)–(4.7) in the form $F \rightarrow G$. We can use Rule (2.4) and (2.5) to simplify them. Whether we use the former or the latter rule depends on the formula to be converted. Let us notice that for any history variable h, such as *open_to*, *ts_sent* and *ts_rcvd*, it is easy to prove that

$$h(i) = j \rightarrow h(i) = j$$

which means that once the value of the variable is non-nil, it will not change any more. Therefore when applying the proof rules (2.4) and (2.5), our goal is to have one of the formulas in the above form.

Applying Rule (2.5) for assertion (B.2) we get:

$$\Box(S.CR[a,b].open\_to[i] = j \Rightarrow C.CR[b,a].open\_to[j] \in \{i, \texttt{empty}\}) \tag{B.6}$$

$$S.CR[a,b].open\_to[i] = j \rightarrow S.CR[a,b].open\_to[i] = j \tag{B.7}$$

Applying Rule (2.5) for assertion (B.3) we get:

$$\Box(C.CR[a,b].open\_to[i] = j \Rightarrow S.CR[b,a].open\_to[j] = i) \tag{B.8}$$

$$C.CR[a,b].open\_to[i] = j \rightarrow C.CR[a,b].open\_to[i] = j \tag{B.9}$$

Applying Rule (2.5) for assertion (B.4) we get:

$$\Box(S.CR[a,b].open\_to[i] = j \wedge k \neq i \Rightarrow S.CR[a,b].open\_to[k] \neq j) \tag{B.10}$$

$$S.CR[a,b].open\_to[i] = j \rightarrow S.CR[a,b].open\_to[i] = j \tag{B.11}$$

Applying Rule (2.4) for assertion (4.5) we get:

$$\begin{aligned}&\Box(R \in C_{a,b} \wedge R.lin = l \wedge R.ts = t \Rightarrow \\ &\quad\Rightarrow\ \ l \neq \texttt{empty} \wedge t \neq \texttt{empty} \wedge C.CR[a,b].ts\_sent[l] = t)\end{aligned} \tag{B.12}$$

$$C.CR[a,b].ts\_sent[l] = t \rightarrow C.CR[a,b].ts\_sent[l] = t \tag{B.13}$$

Notice that here we used a pedantic form of (B.12) in order to make it easier to understand how the proof rule was applied. Applying Rule (2.4) for assertion (4.6) we get:

$$\begin{aligned}&\Box(S.CR[a,b].open\_to[i] = j \Rightarrow \\ &\quad\Rightarrow\ \ S.CR[a,b].ts\_rcvd[i] = C.CR[b,a].ts\_sent[j] \neq \texttt{empty})\end{aligned} \tag{B.14}$$

$$S.CR[a,b].open\_to[i] = j \rightarrow S.CR[a,b].open\_to[i] = j \tag{B.15}$$

Applying Rule (2.4) for assertion (4.7) we get:

$$\begin{aligned}&\Box(A \in C_{a,b} \wedge A.lin = l \wedge A.rin = r \wedge A.ts = t \Rightarrow \\ &\quad\Rightarrow\ \ (l \neq \texttt{empty} \wedge r \neq \texttt{empty} \wedge t \neq \texttt{empty} \wedge S.CR[a,b].open\_to[l] = r \wedge \\ &\qquad\wedge S.CR[a,b].ts\_rcvd[l] = t))\end{aligned} \tag{B.16}$$

$$S.CR[a,b].open\_to[l] = r \rightarrow S.CR[a,b].open\_to[l] = r \tag{B.17}$$

$$S.CR[a,b].ts\_sent[l] = t \rightarrow S.CR[a,b].ts\_sent[l] = t \tag{B.18}$$

Similarly to (B.12), a pedantic version of (B.16) is used to make the application of the proof rule clear. Note that from the above assertions (B.7), (B.11), (B.15) and (B.17) are equivalent.

Finally, here are further assertions needed for the verification. These assertions were generated as necessary preconditions while trying to prove the invariance of the already introduced assertions.

$$\Box(0 \leq i \leq C.CR[a,b].lin \Rightarrow C.CR[a,b].ts\_sent[i] \neq \texttt{empty}) \tag{B.19}$$

$$\Box(i > C.CR[a,b].lin \Rightarrow C.CR[a,b].ts\_sent[i] = \texttt{empty}) \tag{B.20}$$

$$\Box(0 \leq i \leq S.CR[a,b].lin \Rightarrow S.CR[a,b].ts\_rcvd[i] \neq \texttt{empty}) \tag{B.21}$$

$$\Box(i > S.CR[a,b].lin \Rightarrow S.CR[a,b].ts\_rcvd[i] = \texttt{empty}) \tag{B.22}$$

$$\begin{aligned}&\Box(C.CR[a,b].status = \texttt{opening} \Rightarrow \\ &\quad \Rightarrow \quad C.CR[a,b].open\_to[C.CR[a,b].lin] = \texttt{empty})\end{aligned} \tag{B.23}$$

$$\Box(i > C.CR[a,b].lin \Rightarrow C.CR[a,b].open\_to[i] = \texttt{empty}) \tag{B.24}$$

$$\Box(i > S.CR[a,b].lin \Rightarrow S.CR[a,b].open\_to[i] = \texttt{empty}) \tag{B.25}$$

By now we have enough assertions to prove that these are indeed invariants of SCMP. Since some of the assertions listed above are equivalent, here comes the list of assertions whose invariance has to be shown: (B.6), (B.8), (B.10), (B.12), (B.14), (B.16), (4.8), (4.9), (4.10), (4.11), (4.12), (4.13), (B.19), (B.20), (B.21), (B.22), (B.9), (B.13), (B.7), (B.18), (B.23), (B.24) and (B.25).

All of these invariants are either in the form of $P \rightarrow P$ or $\Box P$, where P is a pure predicate (no temporal operators). For the assertions in the first form we have to prove that $\{P\}e\{P\}$ is true for every event of the system as follows from proof rule (2.1). For invariants in the second form, besides proving $P \rightarrow P$, we also have to prove that P is true in any initial state of the system (see proof rule (2.2). By substitution it is easy to check that for all invariants of the second form, the predicate P is true in all initial states.

Let P_i denote the predicate of assertion (i). To show that event e satisfies $\{P_i\}e\{P_i\}$, we can use proof rule (2.3) and define a subset J of the assertions assumed to be invariant such that their conjunction forms a sufficient precondition of P_i with respect to e. That is, instead of proving $\{P_i\}e\{P_i\}$ we only have to prove the weaker $\{\bigwedge_{j\in J} P_j\}e\{P_i\}$.

The assertions listed above are preserved by every event of the system.

B.3 Proof of Lemma 4.4

Here we prove that the modified SCMP specification (given in Section 4.2.7) satisfies the CI requirements (4.27)–(4.32). As we mentioned in Section 4.2.10, the invariance of some of these assertions can be proven without exploiting the real-time properties of the system. We start our proof with these assertions (Appendix B.3.1).

Each of the assertions are in the form $\Box P_i$, where each P_i is a pure predicate. The invariance of such assertions can be proven by using the proof rule (2.2) from Chapter 2. It is easy to verify that each P_i is satisfied in the initial states. What remains to prove is that the Hoare-triple $\{P_i\}e\{P_i\}$ is satisfied by every event e.

B.3.1 Proof without real-time assumptions

Assertion (4.27), repeated below, is affected by the events *C.ClkTick*, *C.Open*, *C.RecvPkt* and *C.Close*.

$$\Box(C.CR[a,b].status \neq \texttt{closed} \Rightarrow \\ \Rightarrow \quad C.CR[a,b].time \geq C.CR[a,b].ts > C.CR[a,b].time - W_C)$$

C.ClkTick preserves the assertion because it increments *C.time* and explicitly checks for CRs records whose timestamp would become lower than $C.time - W_C$. The *C.Open* event sets the consequent of the implication to true ($C.CR[a,b].time = C.CR[a,b].ts$), thus it cannot falsify the assertion. *C.RecvPkt* may change $C.CR[a,b].status$, but only if it equals `opening`. In that case, however, the truth of (4.27) before the event assures that it will be satisfied after the event as well. Finally, *C.Close* changes the antecedent of the implication to false, which implies the truth of the implication.

Assertion (4.29), repeated below, is affected by the events *S.ClkTick*, *C.RecvPkt* and *S.Close*.

$$\Box(S.CR[a,b].status \neq \texttt{closed} \Rightarrow S.time + \epsilon \geq S.CR[a,b].ts\_srvr > S.time - W_S)$$

The cases of *S.ClkTick* and *S.Close* are analog to the cases for the client above. In case of *S.RecvPkt*, we have to show that $R.ts \in_N (l, S.time + \epsilon]$ implies that $R.ts\_srvr = l + ((R.time - l) \bmod N)$ satisfies (4.29). Because (4.29) and (4.30) are true before the execution of the event, we know that $S.time - W_S \leq l \leq S.time + \epsilon$ holds. Therefore, it is enough to show that $0 < (R.ts - l) \bmod N \leq S.time - l + \epsilon$ holds after the event. Substituting these values into the specification of $\in_N$ in Section 4.2.7, we get that the assertion is preserved by this event.

Assertion (4.30), repeated below, is affected by the events *S.ClkTick* and *S.Close*.

$$\Box(S.time + \epsilon \geq S.upper \geq S.time - W_S)$$

In case of *S.ClkTick* the assignment $upper := \max(upper, time - W_S)$ assures that the assertion remains valid after the event. In case of *S.Close*, the truth of assertions (4.29) and (4.30) before the event guarantees that (4.30) remains true.

$$\Box(A \in C_{b,a} \wedge C.CR[a,b].status = \texttt{opening} \Rightarrow$$
$$\Rightarrow \quad C.CR[a,b].ts \geq A.ts \geq C.CR[a,b].ts - N + 1)$$

B.3.2 Proof with real-time assumptions

In order to prove the invariance of assertions (4.28), (4.31) and (4.32), we have to exploit the real-time properties of the system. The assumption about the maximum packet lifetime in the channels, assertion (2.13) and the assumptions about the minimum and maximum interval between clock ticks, assertions (2.14) and (2.15) will be used here. We will list the necessary assertions for the proof, but we will not prove the truth of each Hoare-triple.

$$\Box(R \in C_{a,b} \Rightarrow C.t[R.ts] \leq R.t) \tag{B.26}$$

$$\Box(R \in C_{a,b} \Rightarrow (R.t \leq C.t[R.ts + W_C] \vee C.t[R.ts + W_C] = \lambda)) \tag{B.27}$$

The assertions (B.26) and (B.27) reflect the operation of the client and create the link between the timestamps in requests in the channels and the time of the client which produced these requests. The truth of these assertions is based on the property (see assertion (4.27)) that *C.time* is at least *R.ts* and less than $R.ts + W_C$ when a request is sent.

$$\Box(C.t[S.CR[a,b].ts] \leq S.t[S.CR[a,b].atime + 1] \vee$$
$$\vee \quad S.t[S.CR[a,b].atime + 1] = \lambda) \tag{B.28}$$

$$\Box(S.t[S.CR[a,b].atime] \leq C.t[S.CR[a,b].ts + W_C] + L \vee$$
$$\vee \quad C.t[S.CR[a,b].ts + W_C] = \lambda) \tag{B.29}$$

The assertions (B.28) and (B.29) follow from the previous two assertions, the assumption (2.13) and the definition of the *S.RecvPkt* event. Assertion (B.28) uses (B.26) and

the fact that a packet is received not earlier than it is sent. Assertion (B.29) uses (B.27) and the assumption that a packet stays at most L seconds in the channel before it is delivered.

$$\Box((R \in C_{a,b} \wedge R.ts < C.CR[a,b].ts) \Rightarrow \\ \Rightarrow \quad (R.t \leq C.t[C.CR[a,b].ts + 1] \vee C.t[C.CR[a,b].ts + 1] = \lambda)) \tag{B.30}$$

$$\Box((R_1, R_2 \in C_{a,b} \wedge R_1.ts < R_2.ts) \Rightarrow \\ \Rightarrow \quad (R_1.t \leq C.t[R_2.ts + 1] \vee C.t[R_2.ts + 1] = \lambda)) \tag{B.31}$$

$$\Box((R \in C_{a,b} \wedge R.ts < S.CR[b,a].ts) \Rightarrow \\ \Rightarrow \quad (R.t \leq C.t[S.CR[b,a].ts + 1] \vee C.t[S.CR[b,a].ts + 1] = \lambda)) \tag{B.32}$$

The assertions (B.30)–(B.32) reflect that incarnations do not overlap. That is, when the client starts a new incarnation it will no more resend the request of older incarnations.

$$\Box(S.CR[a,b].atime - W_S < S.CR[a,b].ts\_srvr \leq S.CR[a,b].atime + \epsilon) \tag{B.33}$$

Assertion (B.33) states the fact that when the last request was accepted, its timestamp must have been in the server's imaginary window. The truth of the assertion follows from assertions (4.29) and (4.30) and from the specification of the *S.RecvPkt* event.

Now we have enough information to prove the invariance of (4.31) and (4.32). Our goal is to find a lower bound on $R.ts$ of old packets that may be still in the channel and then to provide a sufficient constraint which assures that old requests cannot be erroneously accepted.

From the clock-rate assumption (2.14), we get (B.34).

$$\Box((S.time - S.CR[a,b].atime - 1) \cdot \gamma \leq S.t[S.time] - S.t[S.CR[a,b].atime + 1] \vee \\ \vee \quad S.t[S.CR[a,b].atime + 1] = \lambda) \tag{B.34}$$

Combining (B.28) and (B.34) yields (B.35).

$$\Box(C.t[S.CR[a,b].ts] \leq S.t[S.time] - (S.time - S.CR[a,b].atime - 1) \cdot \gamma \vee \\ \vee \quad S.t[S.CR[a,b].atime + 1] = \lambda) \tag{B.35}$$

Assertion (B.36) formulates a simple property of the epoch variables $S.t$ which record the clock tick events.

$$\Box(\lambda \neq S.t[S.time] \leq \tau) \tag{B.36}$$

The combination of (B.35) and (B.36) results in (B.37).

$$\Box((C.t[S.CR[a,b].ts] \leq \tau - (S.time - S.CR[a,b].atime - 1) \cdot \gamma) \vee \\ \vee \quad S.t[S.CR[a,b].atime + 1] = \lambda) \tag{B.37}$$

Let us now combine (B.27) with the clock-rate assumption (2.14) which results in (B.38).

$$\Box(R \in C_{a,b} \Rightarrow (\tau \leq C.t[R.ts + W_C] + L \vee C.t[R.ts + W_C] = \lambda)) \tag{B.38}$$

Combining (B.37) and (B.38), we get (B.39).

$$\begin{aligned} &\Box(R \in C_{a,b} \Rightarrow \\ &\quad \Rightarrow \quad (C.t[S.CR[b,a].ts] + (S.time - S.CR[b,a].atime - 1) \cdot \gamma \leq \\ &\qquad\quad \leq C.t[R.ts + W_C] + L \vee \\ &\qquad \vee S.t[S.CR[b,a].atime + 1] = \lambda \vee C.t[R.ts + W_C] = \lambda)) \end{aligned} \tag{B.39}$$

Rearranging (B.39) and using the clock-rate assumption (2.14), we get (B.40).

$$\begin{aligned} &\Box(R \in C_{a,b} \Rightarrow \\ &\quad \Rightarrow \quad S.CR[b,a].ts - R.ts \leq \frac{L}{\gamma} + W_C - (S.time - S.CR[b,a].atime - 1)) \end{aligned} \tag{B.40}$$

Assertion (B.40) is important because it gives a lower bound of the timestamp of any request in the channel. In order to prove the invariance of the assertions (4.31) and (4.32), we will show that any request having a timestamp older than the latest accepted, fails the modulo-N test of the *S.RecvPkt* event. Using the assertions (4.29), (4.30) and the definition of the $\in_N$ relation, we get that (B.41) is a sufficient condition for the truth of (4.31) and (4.32).

$$\begin{aligned}&\Box(R \in C_{a,b} \Rightarrow \\ &\quad \Rightarrow \quad S.CR[b,a].ts\_srvr - (S.CR[b,a].ts - R.ts) > S.time + \epsilon - N)\end{aligned} \tag{B.41}$$

Rearranging (B.41) and using assertions (B.33), (B.40) we get assertion (B.42).

$$\begin{aligned}&\Box(R \in C_{a,b} \Rightarrow S.CR[b,a].ts\_srvr - (S.CR[b,a].ts - R.ts) > \\ &\quad > \quad S.time - W_C - W_S - \frac{L}{\gamma} - 1)\end{aligned} \tag{B.42}$$

Therefore, assertion (B.43) is a sufficient condition for the truth of (B.41). Rearranging assertion (B.43) we get the real-time constraint (4.34).

$$\Box(S.time - W_C - W_S - \frac{L}{\gamma} - 1 \geq S.time + \epsilon - N) \tag{B.43}$$

We still have to prove the invariance of assertion (4.28). Instead of (4.28), we prove a somewhat stronger statement (B.44).

$$\Box(A \in C_{a,b} \Rightarrow A.ts \geq C.time - N + 1) \tag{B.44}$$

Combining assertions (4.29) and (B.33), we get (B.45).

$$\Box(S.CR[a,b].status = \texttt{open} \Rightarrow S.time - S.CR[a,b].atime < W_S + \epsilon) \tag{B.45}$$

Using the assertions (4.7), (B.29), (B.45) and the clock-rate assumption (2.15) we get (B.46).

$$\Box(A \in C_{a,b} \Rightarrow A.t \leq C.t[A.ts + W_C] + L + (W_S + \epsilon) \cdot \Gamma) \tag{B.46}$$

Combining this with the maximum packet-lifetime assumption (2.13) and then with the clock-rate assumption (2.14), we get (B.47) and then (B.48).

$$\Box(A \in C_{a,b} \Rightarrow C.t[C.time] \leq \tau \leq C.t[A.ts + W_C] + 2L + (W_S + \epsilon) \cdot \Gamma) \tag{B.47}$$

$$\Box(A \in C_{a,b} \Rightarrow C.time - A.ts \leq \frac{2L + (W_S + \epsilon) \cdot \Gamma}{\gamma} + W_C) \tag{B.48}$$

From assertion (B.48) we can obtain a sufficient constraint (B.49) for the truth of (B.44). Notice that (B.49) is equivalent with the real-time constraint (4.33).

$$\Box(A \in C_{a,b} \Rightarrow \frac{2L + (W_S + \epsilon) \cdot \Gamma}{\gamma} + W_C \leq N - 1 \tag{B.49}$$

This completes our proof.

Curriculum Vitae

November 28, 1966	born in Budapest, Hungary
1981 – 1985	grammar school I. István Gimnázium, Budapest
September 1985 – August 1986	military service
September 1986 – October 1991	undergraduate student Faculty of Electrical Engineering Technical University of Budapest
September 1991 – April 1992	software engineer RAIR Kft., Budapest
May 1992 – April 1996	Ph.D. student Tele-Informatics and Open Systems Group University of Twente, Enschede
May 1996 – present	researcher Traffic Analysis and Network Performance Lab. Ericsson, Hungary