Verifying the distributed real-time network protocol RTnet using Uppaal[*]

Ferdy Hanssen, Angelika Mader, Pierre G. Jansen
Distributed and Embedded Systems group
Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente
PO-Box 217, 7500 AE, Enschede, the Netherlands, FAX: +31 53 489 4590
E-mail: hanssen@cs.utwente.nl, a.h.mader@utwente.nl, p.g.jansen@utwente.nl

Abstract

RTnet is a distributed real-time network protocol for fully-connected local area networks with a broadcast capability. It supports streaming real-time and non-real-time traffic and on-the-fly addition and removal of network nodes. This paper presents a formal analysis of RTnet using the model checker Uppaal. Besides normal protocol behaviour, the analysis focuses on the fault-handling properties of RTnet, in particular recovery after packet loss. Both qualitative and quantitative properties are presented, together with the verification results and conclusions about the robustness of RTnet.

1. Introduction

This paper describes a formal analysis of the RTnet protocol [1]. It is a typical example of a protocol to be used for streaming data in a distributed environment. These protocols support the transfer of multimedia content and require timely arrival to avoid disturbances of audio or video streams. As a consequence these protocols require real-time behaviour.

The techniques as presented in this paper can be used for an extensive number of streaming data protocols, be it over the net, for a distributed home environment, or currently also for a *network on a chip* [2]. We will illustrate our used technique on basis of the RTnet protocol for which we have built a prototype environment, i.e., the hardware, distributed system software and some test applications.

Our RTnet protocol is to be used on fully connected local area networks with a broadcast capability. Its main goal lies in the combination of Quality of Service (QoS) and flexibility with regard to dynamics in the number of nodes and real-time streams. The basic idea of the RTnet protocol is quite simple. However, distribution and fault handling lead to complexity in the number of possible scenarios that make formal analysis a suitable approach to increase confidence in the correctness of the protocol. We apply model checking as an experimental approach [3]: formal analysis can only increase *trust* in the correctness of an implementation, but not guarantee its ultimate correctness. We are aware of the fact that a formal correctness proof is only about a model, and not about the implementation. Also, we prove protocols only for a fixed number of nodes, not the general case.

In order to get meaningful results from model checking we follow two lines. First, we perform a high number of model checking experiments. In the work presented here, we did approximately 190000 experiments, each of which is determined by the choice of representative values for parameters, such as timing parameters, possible package loss, and different stream sets. The variation on parameters of the model results in 10800 different instances of the protocol model. Each instance is verified against 18 different properties, e.g., occurrence of collision, recovery, and recovery duration. Second, the quality of analysis results can only be as good as the quality of the models used for analysis [4]. A systematic derivation of the model, where assumptions and design decisions are made explicit makes a model more understandable and increases the trust that the model indeed reflects the protocol implementation. Therefore, we clearly state what aspects of the protocol are described in the protocol model.

For the analysis we used the model checker Uppaal [5]. There are three reasons for our choice:

Timed analysis: Uppaal is suitable for verification of real-time systems that can be modelled as timed automata, i.e., non-deterministic processes with a finite control structure and real-valued clocks. These processes may communicate through shared variables or a handshake synchronization. Typical applications of Uppaal are real-time controllers and communication protocols in particular where timing aspects are critical [6–10].

Modelling and analysis support: Uppaal contains a simulator that is useful during the development of the model. The verifier has the possibility to create a diagnostic trace, i.e., a sequence of actions leading to a state where the property of interest fails. Such a sequence can be replayed in the simulator to investigate the scenario leading to the failure.

[*]This work is sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant number 612.060.111, and this work is supported by the IBM Equinox programme.

A straightforward mapping: RTnet is defined using a deterministic state-transition diagram for every node. The class of models that Uppaal can handle, consists of timed, non-deterministic automata and conversion of the deterministic state-transition diagram of RTnet to a Uppaal automaton is relatively straightforward.

The organization of this paper is as follows. Section 2 briefly describes the RTnet protocol. The model of RTnet and the properties used for analysis are described in sections 3 and 4, respectively. We describe the experiments in section 5 and their results in section 6. Finally conclusions are drawn in section 7.

2. RTnet

We briefly describe the distributed RTnet network protocol. An extensive description was published earlier [1].

RTnet can be used in any local area environment where a distributed protocol is needed and where QoS support and flexibility is required. It is a distributed protocol for use on digital networks with real-time requirements. It allows processors connected by a broadcast-capable network to communicate in real-time. The control is distributed and all nodes in the network are expected to cooperate for guaranteeing real-time behaviour.

The protocol requires a fully-connected medium, such as a common bus or Ethernet, where every message sent can be received directly by all other nodes. It supports both real-time and non-real-time communication. Applications can reserve a portion of the available bandwidth to transmit real-time data, organised in streams. Simultaneously, the network can use the remaining bandwidth for best-effort traffic. The key idea is that the network is scheduled dynamically, using standard real-time scheduling algorithms normally used for task scheduling on a single CPU.

RTnet is designed to satisfy a set of goals. These are:

Ability to run on resource-lean devices: it must be implementable on devices with few resources.

QoS guarantees: bandwidth reservations for real-time data must be met, existing streams may not be hindered by newly added streams.

Non-real-time traffic: besides periodic real-time traffic our network protocol should allow for best-effort traffic, which may not affect the deadlines of real-time streams.

Fault tolerance: the physical layer of a network generally does not provide faultless delivery of all packets. Our network protocol should recover from network faults during token transmissions.

Plug-and-play: addition and removal of nodes should be automatic.

Can be based on existing hardware and protocols: it must be possible to use existing hardware and software.

We distinguish two kinds of network traffic: real-time and non-real-time. Real-time data has precedence over non-real-time data, which means that non-real-time data is sent only when no real-time data is being offered by any node. Real-time data is allowed in the form of real-time streams. Each stream is characterised by the bandwidth it needs and a period. Admission control, based on feasibility analysis, is required before admitting a new real-time stream to the network. The source node of a stream is then periodically granted access to the network for that stream.

RTnet is based on a timed token on a broadcast capable network. In normal operation, the token holder may use the network exclusively. The passage of the token through the network does not follow a static, predefined path, but is scheduled on-the-fly by a dynamic version of a real-time scheduler, such as Rate Monotonic (RM), Deadline Monotonic (DM), or Earliest Deadline First (EDF).

There are two variants of RTnet. In *unicast token RTnet* the token is sent to a single receiver, in *broadcast token RTnet* the token is broadcast. The advantage of the broadcast version is reduction of packets that have to be sent. But unicast token RTnet does not need broadcasts for the basic protocol operation, only for on-line node additions. Some network architectures have an inherently lower priority for broadcast packets than for unicast packets: a disadvantage for broadcast token RTnet.

An important aspect of the token is that it does not follow a predefined ring of all nodes. The token follows the path of nodes with the highest priority to use the network.

All information necessary to decide which stream has the highest priority is present in the token. This enables a distributed approach to schedule the streams. All nodes are assumed to collaborate on this distributed real-time scheduling policy. Such a distributed scheduler needs synchronized clocks on the various nodes, so all nodes decide upon the same stream to have the highest priority.

Every node follows the state-transition diagram[1] depicted in figure 1. For simplicity, this diagram shows only the basic transitions and not most error-induced transitions.

There are basically three roles for the nodes that participate in RTnet. A node can be *token holder*, *monitor*, or *idle*. Normally there is one token holder and one monitor. The token holder is in charge of the network, and it is the only node allowed to transmit data. The monitor is the guardian of the token holder, and will come into action only when the token is not propagating correctly. In normal operation the monitor is always the previous token holder. The roles of token holder and monitor are not fixed to specific nodes, all nodes take turns on these roles. The idle role maps directly to the *Idle* state, the monitor role maps to the *Monitor* state and the *Poll* state, and the token holder role maps to the *Announce*, *Activate*, *Dispatch*, and *Transmit* state. In the *Off-line* state the node does not have any role.

[1]Figure 1 depicts the unicast token variant, a figure of the broadcast token variant was published in the extensive description of RTnet [1].

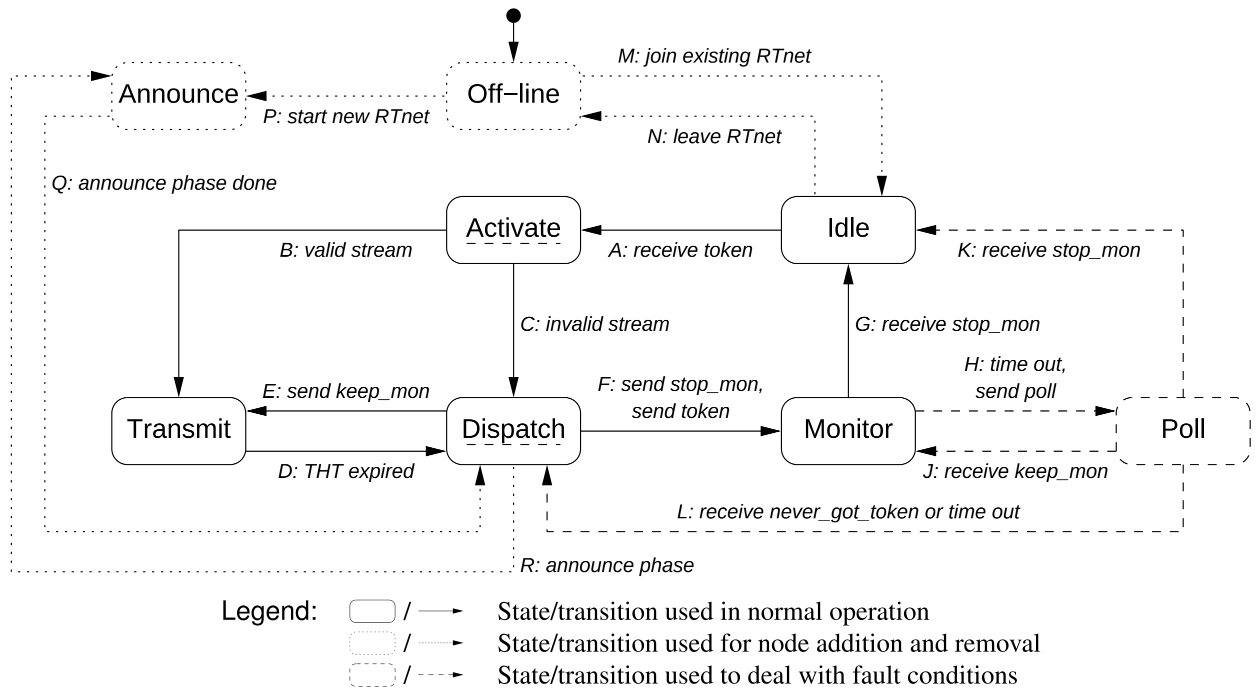

Figure 1. Unicast token RTnet state-transition diagram

Normal operation

In normal operation an idle node will, at one point in time, receive a token (transition *A* in the figure) and assume the role of token holder. The token indicates which stream is selected to begin data transfer. If that stream is a valid stream on that node, it will commence transmission (transition *B*). In the token is also stored for how much time the node is allowed to transmit. If the stream is not valid (e.g., when the responsible application has terminated the stream), the *Transmit* state is skipped and a new stream is selected immediately (transition *C*).

When the data transfer is finished (transition *D*), the next stream is selected. During this computation the node is in the *Dispatch* state. If the next stream originates from the same node, it informs the monitor of this (transition *E*). If the next stream originates from somewhere else, the monitor is informed first and the token is passed to support the next stream (transition *F*). Then the node assumes the role of monitor.

In the *Monitor* state the node will simply wait for an acknowledgement of the token holder that either the token has been passed on (this is called a *stop monitoring* message) or that the token holder would like to keep the token for a bit longer (this is called a *keep monitoring* message). If the token has been passed on, the node will leave the role of monitor and become idle (transition *G*). In the second case it will simply wait for the next acknowledgement. This is the entire cycle when nothing goes wrong.

Fault handling

We assume network packets arrive intact or not at all. They are not modified in transit. We assume that nodes are fail silent, i.e., when they are down they produce no network traffic at all.

When the acknowledgements to the monitor do not arrive in time, the monitor assumes something is amiss. The time out of the monitor is based on the Token-Holding Time (THT) it allocated in the *Dispatch* state, extended by a small margin, called δ_{poll}. The monitor will have to find out what went wrong and take appropriate action. To do this it will first observe the network. If there is activity on the medium from another node than the one it is monitoring, it may conclude the token has not been lost, only a *stop monitoring* message was lost. The node can then stop monitoring and becomes idle.

If there is activity from the node it is monitoring, it may conclude a *keep monitoring* message has been lost. Now the monitor will have to actively monitor the token holder, as it does not know how long it will have to wait until the next *stop monitoring* or *keep monitoring* message arrives.

If there is no network activity, the conclusion is that the token is lost. Now the monitor has to determine whether the prospected token holder is still alive. To do this it will transmit a *poll* message (transition *H*) and wait a short time for an answer in the *Poll* state. The length of this short wait is δ_{poll}, the same as the margin added to the THT before actually sending a *poll* message.

If the token holder is alive and has a valid token, it will reply with a *keep monitoring* message, indicating how long it wants to keep the token and the monitor will restart its time-out timer (transition *J*). If the token holder is alive, but already forwarded the token, it will reply with *stop monitoring*. The monitor will then become idle (transition *K*).

When the monitor receives a reply from the token holder that states that a token has never been seen, the monitor selects a new stream and a new token will be sent (transition *L*). If the token holder does not respond, the monitor will delete it from the list of active nodes, together with all streams originating from it, and a new stream will be selected for transmission (transition *L*).

Dynamic network updates

Periodically the *announce phase* of the network is activated. The token holder enters the *Announce* state (transition *R*): it broadcasts an invitation and waits for replies. After a short time the next stream is selected (transition *Q*).

Nodes may join a live network in response to an invitation asking for new participants and leave it (transitions *M* and *N*). A node that does not receive this invitation within a specified time out may start a new RTnet (transition *P*).

3. The Uppaal model

Timed automata

Systems are modelled in Uppaal as a parallel composition of non-deterministic, timed automata. Time is modelled using real-valued clocks and time only progresses in the locations of the automata: transitions are instantaneous. The guards on transitions between the various locations in the automata and the invariants in the various locations may contain both integer-valued variables and real-valued clocks. Clocks may also be reset to some constant in the transitions. Several automata may also synchronize on transitions using handshakes. With the use of shared variables it is possible to model data transfer between automata. Locations may be urgent, which means time is not allowed to progress, and committed, which means time is not allowed to progress and interleaving is restricted. If only one automaton is in a committed location at any one time, its transitions are guaranteed to be atomic.

Properties of systems are checked by the Uppaal model checker, which performs an exhaustive search through the state space of the system for the validity of these properties. It can check for invariant, reachability, and liveness properties of the system. The properties are expressed using a subset of Timed Computational Tree Logic (TCTL) [11]. Table 1 summarises the forms needed for our properties.

Table 1. Uppaal property forms

Syntax*	Meaning
$\exists\Diamond\ p$	Possibly p holds, i.e., there is a state where p holds that is reachable from the initial state.
$\forall\Box\ p$	Invariantly p holds.
$p \leadsto q$	Whenever p holds, eventually q will hold too.

* p and q are combinatorial, boolean expressions of automaton locations and relational, numeric expressions over constants, variables, and clocks

Modelling the RTnet protocol

We first describe a number of modelling decisions.

Fragment: we model the basic protocol consisting of the normal operation mode and the fault handling mode. This corresponds to the lower six states (with respective transitions) in figure 1. *Not* in the model are the announcements (adding new nodes), the off-line phase, and also the dynamic addition of streams. Reason for that choice were complexity issues.

Protocol and model variants: the protocol exists in two variants: unicast and broadcast token. The differences between these two variants do not seem very large on a high-level perspective. But the differences in the details, especially fault handling, are significant enough to warrant the use of different models for the two variants.

Decomposition in automata: there is basically an automaton for each node, and an automaton for the network. For more complex verification properties we needed a second network automaton to identify specific states. There are two kinds of network communication: control packets and stream data packets. Control packet transmission is modelled by handshake synchronisation, between sending node and network, and another one between network and receiving node. The network automaton contains a non-deterministic choice whether a packet gets lost or not. Stream data packet transmission is modelled implicitly by only letting time pass.

Node automata. The node automaton corresponds directly to the protocol state-transition diagram, shown in figure 1. A node is always ready to synchronize on one of its receive channels, i.e., in all non-committed locations it accepts an RTnet control message. In a real system a node may, at any time, receive a control message, due to faults. When this message is unexpected a node will disconnect itself from RTnet and try to join again later. In the model the node automaton will enter an error location upon receipt of an unexpected message and not participate in the protocol any more, thus modelling the automatic disconnection.

Data transfer, i.e., network traffic that is application-specific data and not an RTnet control message, is not modelled explicitly, but it is assumed that data is transferred while time passes in the *TRANSMIT* location. Loss of data packets is not modelled, because the RTnet protocol is not concerned with the correctness of application data and not designed to be. It is assumed the protocol stack on top of RTnet or the application will handle data corruption. The main objective of the model is to verify if the protocol works when control messages are lost, because these are essential to the proper operation of RTnet.

Since non-real-time data handling is outside the scope of this analysis, it is modelled in a simplistic way. In RTnet the token rotates in a Round Robin (RR) fashion among the nodes to allow for a fair share of time for each node to transmit its non-real-time messages. In the model this RR behaviour is not modelled. Instead, if no real-time stream is ready to send, a special stream is selected, called the *best-effort stream*. This stream has no source node: if this stream is selected the token remains at the node that sent the last real-time stream. The token is only forwarded again when the next real-time stream becomes ready to send.

Network automata. The network is essentially modelled with a single automaton. This automaton waits for a handshake synchronization with a sending node. There is a different handshake channel for each control message type, allowing for easy control message identification at the receiving node and simple differentiation between message types in the network automaton, as not all control messages have to be handled in the same way.

After the handshake synchronization with the sending node, the network automaton non-deterministically either performs a handshake synchronization with the receiving node or it waits for a new handshake with a sending node. In the latter case the control message is lost. Note that the network automaton does not introduce a delay on control messages. This is reasonable because the network delay is on a time scale one order of magnitude finer than the time effects investigated with our model.

A message will arrive at the receiving node at the same time as it left the sending node, but a message is guaranteed to arrive at the receiving node at a later point in the sequence of events than it left the sending node. Also note that the network automaton will always process only one control message type at the same time. In certain configurations it is possible that multiple *poll* messages are processed at the same moment. If more than one *poll* message is processed at the same moment, these are, by definition, all lost.

There are two extended versions of the network automaton. These are needed for detection of multiple independent, simultaneous control messages that are sent at the same time. Such multiple independent, simultaneous control messages might, on a real network, collide and cause non-deterministic transmission delays or delivery failures. The extended network automata detect invalid successions of control messages.

4. Verification properties

We want to analyse the following aspects of RTnet:
1) normal operation,
2) recovery from network failures and recovery time, and
3) occurrences of simultaneous control messages.

In total, there are 18 different properties that allow conclusions about the issues above. They are verified using Uppaal for different versions and configurations of our protocol model. Therefore, they have to be expressed in the Uppaal property language. In the following we will discuss the most important properties.

Properties to analyse normal operation

A state of the model is called *stable* when the following conditions are met:
- at most one node has the role of token holder, call this condition $\psi^{\text{1/t/holder}}$;
- at most one node has the role of monitor, call this condition $\psi^{\text{1/monitor}}$;
- no nodes are about to send or have sent a *poll* message, call this condition $\neg\psi_i^{\text{ext/poll}}$ for every node i;
- no nodes are off-line, i.e., are in the location *ERROR*.

We consider the first three stability conditions *strong*: they must always be met. The last stability condition we consider *weak*: a network with one or more off-line nodes can be stable and operating correctly, just with fewer nodes.

Using the definitions of $\psi^{\text{1/t/holder}}$, $\psi^{\text{1/monitor}}$, and $\neg\psi_i^{\text{ext/poll}}$ the stability condition ψ^{stable} can be constructed for a system with n nodes:

$$\psi^{\text{stable}} = \psi^{\text{1/t/holder}} \wedge \psi^{\text{1/monitor}} \wedge \bigwedge_{i=1}^{n} \neg\psi_i^{\text{ext/poll}} \wedge \bigwedge_{i=1}^{n} \neg \mathit{node}_i.\mathit{ERROR}$$

With the condition ψ^{stable} the stability property ϕ^{stable} to verify the model with may be constructed:

$$\phi^{\text{stable}} = \forall\Box\, \psi^{\text{stable}}$$

Property ϕ^{stable} expresses that the system is always stable. The expected result of this property is *true* if the parameter *packet loss* is set to zero and *false* otherwise.

Other properties whose definitions will not be shown due to space constraints, provide useful insights in the normal behaviour of the protocol:
- deadline misses are checked by the property $\phi^{\text{dl/miss}}$;
- $\phi^{\text{net/idle}}$ expresses if the best-effort stream is ever used, i.e., if there is room for non-real-time traffic;
- $\phi^{\text{1/transmit}}$ expresses that there is at most one node transmitting, i.e., is in the *Transmit* state of the protocol;
- $\phi^{\text{1/dispatch}}$ expresses that there is not more than one node computing the next stream, i.e., is in the *Dispatch* state;
- $\phi^{\text{1/monitor}}$ expresses that there is at most one monitor, i.e., at most one node is in the *Monitor* state of the protocol.

Properties to analyse network recovery

Network failure recovery basically comes down to the question if packet loss results in a permanently stable state of the system. When a packet is lost, the system may reach an unstable state, e.g., because the monitor sent a *poll* message to some other node to actively recover from a fault, and thus the third stability condition is violated.

Property $\phi^{\text{stabilize}}$ expresses that an unstable state of the system will always lead to a stable state of the system, i.e., if the system will recover from a fault. We expect $\phi^{\text{stabilize}}$ to be satisfied if at most one packet loss is allowed (for the network automaton), and possibly *true* otherwise. The stabilization property $\phi^{\text{stabilize}}$ is constructed using ψ^{stable}:

$$\phi^{\text{stabilize}} = \neg\psi^{\text{stable}} \rightsquigarrow \psi^{\text{stable}}$$

For satisfaction of the property $\phi^{\text{stabilize}}$ no node may reach the location *ERROR*. A weaker form of this property, called $\overset{\star}{\phi}{}^{\text{stabilize}}$, allows nodes to reach the location *ERROR*, i.e., the fourth stability condition has been dropped. With this property we want to check, if the protocol, even in the case of errors and nodes being off-line, still continues working with the remaining nodes.

Another interesting property is whether there is at most one node trying to actively recover from a fault. Property $\phi^{1/\text{poll}}$ expresses whether at most one node is in the protocol's *Poll* state. This property is expected to be satisfied with a packet loss of upto one, but may not be satisfied with higher packet loss.

Properties to analyse recovery duration

We define the recovery duration to be the time between a packet loss and the moment of recovery. We determine the possible recovery durations by finding the *minimal* and the *maximal* duration for a given set of parameters. Details are provided in appendix A.

The maximal duration is estimated to be the sum of the estimated largest continuous network usage C_{LCU} of a single stream and one poll delay. For unicast token RTnet we expect this estimate to be the largest time needed before the recovery procedure is started in the case of single packet loss. For broadcast token RTnet a single token loss equals the loss of two packets in the unicast token variant, a *stop monitoring* message with a subsequent *token* message. In that case this estimate may be too low and actually is in some cases, as we will see in section 6.

Properties to analyse occurrence of simultaneous control messages

A faulty situation that could occur is when at least two nodes send a control message at the same moment. To evaluate if this situation of multiple independent, simultaneous control messages can occur, two properties are needed. The first property, $\phi^{\substack{\text{simultaneous}\\ \text{packets sent}}}$, expresses whether this situation occurs. This property can only be used in conjunction with the extended network automata.

There is only one asynchronous control message present in the RTnet protocol: the *poll* message. Multiple independent, simultaneous control messages should only be caused by multiple nodes sending a *poll* message at the same time. To evaluate if this is indeed the case, property $\phi^{tp>1}$ is satisfied if more than one node can be ready to send a *poll*.

The results of verifying properties $\phi^{\substack{\text{simultaneous}\\ \text{packets sent}}}$ and $\phi^{tp>1}$ on models using one of the extended network automata makes it possible to derive situations where multiple independent, simultaneous control messages are *not* caused by multiple *poll* messages and to determine their causes and possible solutions.

Table 2. Stream set $\mathcal{S}_C$ (4 nodes, utilization ≈ 82%)

Stream ID	0	1	2	3	4	5	6	7
Source ID	0	1	2	3	0	1	2	2
Period	3000	3000	3000	3000	2000	500	500	250
Workload	10	10	10	10	20	100	200	50

5. Experiments

We performed model checking runs with Uppaal 3.5.9 for 10800 different configurations, and we checked 18 properties for each configuration. Each configuration is determined by the following settings and parameters:

1) **Unicast token** and **broadcast token** variants of RTnet.
2) There are three **network model configurations** that allow for different analysis of situations where packets are sent simultaneously. The simple network automaton cannot detect simultaneously sent packets. Both extended network automata can detect these. The difference between these two automata is that one takes simultaneous polls into consideration, and one does not. The combination of results derived with both extended network automata allows us to reason about the presence of simultaneous packets that are not both polls.
3) **Stream sets**: these sets are aimed at obtaining as many different situations in the models as possible with varying degrees of complexity. Complexity in this case means mostly the difference in stream periods. Usually it can be stated that the larger the least common multiple of the stream periods is, the larger the state space of the model will be. We use 10 different stream sets, two examples are shown in tables 2 and 3.
4) The scheduler works in units of 10 time units, which allows for values of the poll delay both smaller and larger than this scheduling granularity. The values for the **poll delay** δ_{poll} used in the experiments are 4, 5, 6, 10, and 50 time units. There is always a delay of $2\delta_{poll}$ before

Table 3. Stream set $\mathcal{S}_D$ (4 nodes, utilization $\approx$ 96%)

Stream ID	0	1	2	3	4	5
Source ID	0	1	2	0	3	1
Period	300	400	400	600	600	2400
Workload	30	70	70	60	240	20

the monitor will possibly generate a new token. Since the scheduling granularity is 10 time units, a δ_{poll} of 5 time units makes it likely that the generation of a new token coincides with a scheduling decision on another node. The values 4 and 6 time units are used to determine the behaviour of RTnet when using a δ_{poll} around this point. A δ_{poll} of 10 time units means the *poll* message is likely to coincide with the dispatch phase of another node. The value 50 time units is chosen, because the current prototype uses this arbitrarily chosen value.

5) **Packet loss** is varied between 0 and 2 packets. Disabling packet loss is used to check RTnet behaves as it should in a normal situation, *poll* messages should not occur in this case. The loss of one packet is a situation RTnet is designed to handle, which means nothing must go wrong in this case either. *Poll* messages can occur, but the *ERROR* location of a node should never be reached. The loss of two or more packets is a situation RTnet is not designed to handle with minimal disruption. A loss of two packets is used to experiment how well RTnet can deal with this situation.
6) A node may transmit for a shorter duration than allocated: it releases the token earlier than expected. Nondeterministically modelling all possible (earlier) release times for a node leads to an instance of the well-known state-space explosion problem. A modelling approximation is to limit the number of moments when a node may use less time to 0, 1, and 2. We call this parameter **early token release**. (Additionally, built into the model as constants, we allow only two different moments in time when a token may be released earlier.)
7) The situation can occur that two monitoring nodes send a *poll* message at the same moment. There are two possibilities for the resulting situation, depending on the technical properties of the network used: either both *poll* messages cause a collision and both are lost, or they arrive sequentially. The parameter **simultaneous polls** allows to choose if collisions are possible.
8) Observation of the network traffic can give useful information to nodes busy with fault handling. In certain situations a node holding the token has no data packets to send. Other nodes cannot detect that the token-holding node is still alive, if it does not send packets. A solution here is that a token-holding node sends "still-alive" packets periodically. The model parameter **best-effort data transmission** allows to switch on and off the sending of "still-alive" packets when the token holder has selected the best-effort stream. If a real-time stream is selected, a node is assumed to send data packets belonging to that stream, i.e., a real-time stream is assumed to always have data waiting to be sent.

Table 4. Verification results overview with packet loss of 1

RTnet variant	ϕ^{stable}	$\phi^{\text{dl/miss}}$	$\phi^{\text{net/idle}}$	$\phi^{\text{1/transmit}}$	$\phi^{\text{1/dispatch}}$	$\phi^{\text{1/monitor}}$
Unicast token	*false*	varies	mostly *true*	*true*	mostly *true*	*false*
Broadcast token	*false*	varies	mostly *true*	mostly *true*	varies	*false*

RTnet variant	$\phi^{\text{stabilize}}$	$\overset{\star}{\phi}^{\text{stabilize}}$	$\phi^{\text{1/poll}}$		$\phi^{tp > 1}$	$\phi^{\text{simultaneous packets sent}}$
Unicast token	*true*	*true*	*true*		*false*	varies
Broadcast token	*true*	*true*	varies		varies	varies

6. Results

All results described below are based on verifications by the model checker. For most experiments the model checker generated an answer, but a small percentage caused the model checker to run out of memory, mainly in the experiments where more than one packet is lost.

6.1. Protocol behaviour during normal operation

All verification results for normal operation are in agreement with the expected results. For each property the results are the same, regardless of the choice of settings and parameters. An exception is packet loss that is obviously set to zero. Only the result of $\phi^{\text{net/idle}}$ is not the same for all settings and parameters: stream sets with a utilization of 100% do not have space for non-real-time traffic when early token releases are not allowed, and the verification results confirm this. From this we may conclude that the model of the protocol operates properly when no network faults occur.

6.2. Recovery when one packet is lost

In the case of a single packet loss most verification results are as expected. An overview of the properties mentioned in section 4 is shown in table 4. For each protocol variant the results are listed for the most important properties. The term "varies" means diverse results, depending on the parameters and settings. Results printed in italics are the expected results. The anomalies are explained in detail in appendix B, we provide a summary here.

Both unicast and broadcast token RTnet stabilize without any node going off-line, i.e., reaching the *ERROR* location. Deadline misses occur in both variants, but they are to be expected when control packets may be lost.

A design flaw was found during the verification process. Two nodes could reach the *Dispatch* state at the same time, caused by the wrong reply to a *poll* message by an idle node. The solution is to add more information to the *poll* message so the recipient can map the *poll* message to the corresponding *token* and provide a correct reply.

The broadcast token variant, however, is less robust than the unicast token variant. In broadcast token RTnet it is possible that more than one node actually reach the *Transmit* state at the same time. This is caused by a collision between two simultaneous *poll* messages.

Simultaneous transmissions of control messages may occur in both protocol variants in the presence of network faults. One of those control messages is always a *poll* message. These simultaneous transmissions occur less frequently with unicast token than with broadcast token RTnet. The model uses a constant poll delay that is the same for all nodes. The chance that simultaneous transmissions occur is already small, but it can be made smaller by introducing a random delay in the poll delay δ_{poll}. Then the chance of two simultaneous transmissions occurring is negligible, if the random poll delay is combined with a listen phase before deciding to send an actual *poll* message.

6.3. Recovery when more than one packet is lost

When more than one packet loss is allowed, the results deteriorate somewhat. This can be expected. Simultaneous control messages occur more frequently then.

In the absence of early token releases, stabilization always occurs, albeit with nodes reaching an error state in some of the experiments. When early token releases are allowed, the stabilization results are inconclusive, as a considerable amount of these experiments could not be run due to lack of memory in the model checker.

6.4. Recovery duration

The maximum recovery durations are different for each RTnet variant. The minimum recovery duration is 0 time units for unicast token RTnet: the delay between a *stop monitoring* message and the subsequent *token* message. For broadcast token RTnet the minimum recovery duration is either the poll delay δ_{poll} or the scheduling granularity, depending on the stream set used.

The maximum recovery duration for unicast token RTnet is exactly the estimated value $C_{\text{LCU}} + \delta_{poll}$ in approximately 75% of all cases. In all other cases it is lower, the stream set is then the only parameter that influences this.

For broadcast token RTnet the maximum recovery duration is exactly the estimate in approximately 65% of all cases. It is lower than the estimate in approximately 30% of the cases, and higher in the remaining 5%. The reasons for it being higher are twofold.

All are caused by two polls colliding and then being dropped, but the sequence of events after that are different. One sequence of events is due to the time out after sending a *poll* message without getting a reply. In this case the maximum duration is $C_{\text{LCU}} + 2\delta_{poll}$.

The other sequence of events is slightly more complex: a time out occurs after the disappearance of the simultaneous polls, but both nodes then decide that it is time to transmit best-effort traffic. This may only happen when early token releases are allowed and best-effort data transmission is not required. As both tokens are regenerated, both nodes do not have to inform a monitor, and no control messages are exchanged until some real-time stream becomes eligible for transmission. This may cause a very long delay before the recovery is completed, as both nodes cannot detect each other's best-effort data transmissions.

7. Conclusions

We have analysed the RTnet protocol with the timed automaton model checker Uppaal. Variation of parameters for timing and data result in 10800 instances of the model, that each have been verified against 18 different properties. These properties cover qualitative as well as quantitative aspects.

The model checker and its interface have turned out to provide a comfortable environment for modelling and analysis. The expressiveness and structure of timed automata allows for straightforward modelling of RTnet.

We experienced as a drawback that Uppaal does not quantify probabilities: it analyses *all* possible scenarios. Having found a scenario leading to an undesired state, there is no information about the probability of this scenario.

We have provided a model that is *insightful*, therefore this gives confidence that the model reflects the implementation, i.e., it is also *truthful*. These quality criteria of a model, together with the high number of model checking experiments, increase our *trust* that the RTnet protocol behaves as expected.

Moreover, during analysis a design flaw was detected, leading to an improved version of the protocol.

The verification shows that RTnet operates as advertised in the absence of network faults. Unicast token RTnet seems more robust than broadcast token RTnet when network faults are present. Fault recovery may cause stream deadlines to be missed. If this cannot be tolerated, the feasibility analysis that is done at stream admission time should take recovery times into account. We found a good estimate for the maximum recovery time for both RTnet variants.

For better performance RTnet can use early token releases on network architectures with collision detection. But early token releases should be disabled on networks without collision detection, as it disturbs fault recovery.

We have shown that, despite some minor drawbacks, our structured and experimental model checking approach, in combination with Uppaal, allowed us to create a trustful RTnet. We can recommend it for the analysis of other network protocols with real-time or QoS requirements.

References

[1] F. Hanssen, P.G. Jansen, H. Scholten, and S. Mullender, "RTnet: a distributed real-time protocol for broadcast-capable networks," in *Proc. Joint Int. Conf. on Autonomic and Autonomous Systems and Int. Conf. on Networking and Services (ICAS/ICNS 2005)*. IEEE Computer Society Press, Oct. 2005, ISBN 0-7695-2450-8.

[2] N. Kavaldjiev, G.J.M. Smit, P.G. Jansen, and P.T. Wolkotte, "A virtual channel network-on-chip for GT and BE traffic," in *Proc. IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI '06)*. IEEE Computer Society Press, Mar. 2006, pp. 211–216, ISBN 0-7695-2533-4.

[3] E. Brinksma, "Verification is experimentation!" *Int. J. on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 107–111, May 2001.

[4] E. Brinksma and A. Mader, "On verification modelling of embedded systems," Centre for Telematics and Information Technology, University of Twente, Enschede, the Netherlands, Tech. Rep. TR-CTIT-04-03, Jan. 2004. [Online]. Available: http://www.ub.utwente.nl/webdocs/ctit/1/000000e6.pdf

[5] K.G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *Int. J. on Software Tools for Technology Transfer*, vol. 1, no. 1–2, pp. 134–152, Dec. 1997.

[6] P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans, "The Bounded Retransmission Protocol must be on time!" in *Proc. 3rd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, E. Brinksma, Ed., vol. LNCS 1217. Springer-Verlag, Apr. 1997, pp. 416–431, ISBN 3-540-62790-1.

[7] A. David and W. Yi, "Modelling and analysis of a commercial field bus protocol," in *Proc. 12th Euromicro Conf. on Real-Time Systems*. IEEE Computer Society Press, June 2000, pp. 165–172, ISBN 0-7695-0734-4.

[8] K. Havelund, A. Skou, K.G. Larsen, and K. Lund, "Formal modeling and analysis of an audio/video protocol: An industrial case study using Uppaal," in *Proc. 18th IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, Dec. 1997, pp. 2–13, ISBN 0-8186-8268-X.

[9] H.E. Jensen, K.G. Larsen, and A. Skou, "Modelling and analysis of a collision avoidance protocol using SPIN and UPPAAL," in *Proc. 2nd Workshop on the SPIN Verification System*, vol. DIMACS 32. American Mathematical Society, Aug. 1996, pp. 1–20.

[10] H. Lönn and P. Pettersson, "Formal verification of a TDMA protocol start-up mechanism," in *Proc. 1997 Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS '97)*. IEEE Computer Society Press, Dec. 1997, pp. 235–242, ISBN 0-8186-8212-4.

[11] R. Alur, C. Courcoubetis, and D.L. Dill, "Model-checking for probabilistic real-time systems (extended abstract)," in *Proc. 18th Int. Colloquium on Automata, Languages and Programming (ICALP '91)*, J. L. Albert, B. Monien, and M. Rodríguez-Artalejo, Eds., vol. LNCS 510. Springer-Verlag, July 1991, pp. 115–126, ISBN 3-540-54233-7.

Appendix A
Determining recovery duration

To determine the recovery time, the model is supplied with an additional clock and some flags. Upon packet loss, this clock is reset. The lower and upper bounds for the recovery time then equal this clock's lower and upper bounds when the flags are set properly and the system is stable.

This approach only works with a single packet loss, as a second packet loss will reset the clock, thus losing the information about the earlier packet loss. The solution would be that the network automaton detecting the (second) package loss does not reset the clock, if the system is not stable yet. However, stability of the system is a distributed property that the network automaton has no information about in the current setting.

The bounds of the additional clock are determined by iteration. Finding the lower bound for the recovery time is done by evaluating property $\phi^{\text{l/bound}}(t)$ for all values of t, starting at $t = 0$ and evaluating increasing values of t until $\phi^{\text{l/bound}}(t)$ is satisfied.

Similarly the upper bound for the recovery time is found by evaluating property $\phi^{\text{u/bound}}(t)$ for all values of t. The algorithm used to find the upper bound starts at $t = C_{\text{LCU}} + \delta_{poll}$. If $\phi^{\text{u/bound}}(t)$ is *false* for this starting value, $\phi^{\text{u/bound}}(t)$ is evaluated for increasing values of t until $\phi^{\text{u/bound}}(t)$ is *true*, because the estimate is too low. Otherwise $\phi^{\text{u/bound}}(t)$ is evaluated for decreasing values of t until $\phi^{\text{u/bound}}(t)$ is *false*. The estimated largest continuous network usage C_{LCU} of a single stream is given by:

$$C_{\text{LCU}} = \min\left(\max_i\{C_i\}, 2\min_i\{T_i - C_i\}\right)$$

where T_i is the period of stream i and C_i is its workload: $C_i = (B_i/B)T_i)$, where B_i is the bandwidth required by stream i and B is the available bandwidth on the network.

Appendix B
Packet loss recovery and simultaneous control messages

In the case of a single packet loss most verification results are as expected. An overview is shown in table 4 in section 6.2. The anomalies are explained below.

Unicast token RTnet recovery

Unicast token RTnet produces verification results almost as expected. It always stabilizes without throwing out nodes. Deadline misses do occur, but they are to be expected when some nodes do not receive the token in time and the stream set has a high utilization.

***Poll* message reply.** We found the following design flaw. There are a few cases where two nodes are in the *Dispatch* state at the same time. This is caused by a wrong reply to a *poll* message by an idle node. There are two possible responses to this *poll*. An earlier version of RTnet always responded with a *never got token* message. There exists a situation where this reply is wrong, resulting in two token holders. Verifications of an earlier version of the model show this problem. Note that a prerequisite for this situation to occur is that a node is allowed to use less transmission time than allocated and release the token early.

To fix this problem it is necessary to map a *poll* message to the specific token it is referring to. If the token has already been seen, a *stop monitoring* message can be replied to the *poll*, instead of a *never got token* message. To do this the *poll* message has to be augmented with the stream identifier and a counter, to provide the poll to token mapping.

The current model still allows multiple nodes being in the *Dispatch* state because the counter for the poll to token mapping is not large enough. The model uses a 1-bit counter: enough if the poll delay is not too large with respect to the scheduling granularity. The property $\phi^{1/\text{dispatch}}$ is false with a packet loss of one only when δ_{poll} is 50 time units. If a large δ_{poll} is used, or if higher packet loss needs to be dealt with, a larger counter is needed.

Broadcast token RTnet recovery

The broadcast token variant of RTnet is less robust than the unicast token variant. Broadcast token RTnet always stabilizes without any node reaching the *ERROR* location, i.e., without throwing out any nodes. But when simultaneous polls and early token releases are allowed, in 40% of the experiments more than one node reach the protocol's *Dispatch* state. In 15% of these experiments more than one node actually reach the *Transmit* state at the same time.

The reason for more than one node reaching the *Transmit* state is as follows. Two monitors may decide to send a poll at the same time, because the poll delay δ_{poll} is equal for both. When these two simultaneous polls collide and are lost, both polling nodes decide a new token is needed at the same time, again because of equal poll delays.

One node finds it needs to keep the token, while the other node forwards its token. When the node that kept its regenerated token notices the other token being forwarded, there are two possible courses of action for this node: (1) stop transmitting, discard the local token and wait for a new token or announcement to resume normal operation, or (2) send a message to both source and destination of the observed token, telling them to discard that token. Given that all nodes are equal and it is impossible to distinguish which token is the "better one", the first choice is made, as it has the least possibilities of network disruption.

Deadline misses occur in this variant as well, but they are expected, just as with unicast token RTnet. With a high utilization these cannot be avoided when packets are lost.

Simultaneous transmissions

Independent, simultaneous control messages do occur, but there is always at least one *poll* message involved. This is logical as the *poll* message is the only asynchronous message in RTnet. Although a node that is about to send a *poll* message always listens first before deciding a *poll* message is necessary, it is always possible that a *poll* message is transmitted at the same moment as some other message.

The model only describes collisions between simultaneous *poll* messages and drops them. The result is that both monitors that sent the *poll* messages conclude that the intended poll recipients have failed. Both remove them from their local token. As dynamic node additions are not modelled, the nodes that have been removed from the token will remain idle forever. They have no means of detecting they are no longer part of the node list in the token. The real RTnet lets these nodes rejoin at the next announcement phase.

The model deals with combinations of a simultaneous *poll* message and some other control message in a serial fashion. These situations are invariably caused because the monitor that is listening for network traffic, does not observe these messages. In many cases the monitoring node that will eventually send a *poll* message, is already actively observing its token holder. This token holder is transmitting non-real-time traffic. Because the non-real-time phase is not modelled with the token travelling along all nodes in a RR fashion, the monitor will keep observing its token holder until it tries to transfer the token elsewhere. And there are situations that this token transfer is at the same moment in time as the next time out of the monitor, resulting in a superfluous *poll* message.

Simultaneous transmissions are more frequent with the broadcast token variant than with the unicast token variant. But comparing simultaneous transmissions of the broadcast token variant with one packet loss to the unicast token variant with a packet loss of two shows that simultaneous transmissions occur with a comparable frequency. This can be explained by the fact that the complete loss of a broadcast token is equal to the loss of a *stop monitoring* message and its successive *token* message in unicast token RTnet.

For the broadcast token variant simultaneous transmissions are mostly caused by the same reason as they are for the unicast token variant. The additional cause is due to a second token on the network. As discussed in the section on broadcast token RTnet recovery above, this token is observed by the node that is transmitting data, i.e., is in the *Transmit* state. This node will relinquish its token and become idle, waiting for the token to arrive or the next announcement phase, to rejoin the network if it was deleted from the regenerated token.

The model uses a constant poll delay that is the same for all nodes. The chance that simultaneous transmissions occur is already small, but it can be made smaller by introducing a random delay in the poll delay δ_{poll}. Then the chance of two simultaneous transmissions occurring is negligible, if the random poll delay is combined with a listen phase before deciding to send an actual *poll* message.