# RTnet: a distributed real-time protocol for broadcast-capable networks*

Ferdy Hanssen, Pierre G. Jansen, Hans Scholten, Sape Mullender
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

## Abstract

*RTnet is a distributed real-time network protocol, to be used on fully-connected local area networks with a broadcast capability. It supports on-the-fly addition and removal of network nodes, resource-lavish and resource-lean devices, streaming real-time and regular non-real-time traffic. It allows the use of standard real-time scheduling paradigms to control network traffic, allows dynamic scheduling and is flexible in stream handling. The design is presented, together with measurement results of an experiment with an implementation on top of Ethernet.*

## 1. Introduction

RTnet 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 to guarantee real-time behaviour. We describe the goals of the protocol, its design and operation, and the analysis of an implementation on top of Ethernet.

The protocol requires a fully-connected communication medium, such as a common bus, Ethernet, or a radio medium, where every message sent can be received directly by all other nodes. Broadcast capabilities are needed to enable automatic addition and removal of nodes and to synchronize the nodes' clocks.

The protocol supports both real-time and non-real-time communication. Applications are able to reserve a portion of the available bandwidth to transmit real-time data, organised in streams. At the same time the network uses the remaining bandwidth for best-effort traffic. The network is scheduled dynamically, using standard real-time scheduling algorithms normally used to schedule tasks on a CPU. Nodes and streams may be flexibly added and removed.

The protocol can be used in any local area environment where a distributed protocol is appropriate and where Quality of Service (QoS) support and flexibility is required. We will show the analysis and the performance of a current prototype on top of Ethernet.

*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.

## 2. Goals of the protocol

The requirements and constraints of our network are:

*Able to run on resource-lean devices:* it must be implementable on devices with both few and many resources.

*QoS guarantees:* to prevent buffer under- and overflow the network should guarantee that messages are delivered before their deadlines. Thus a scheduling technique is required to assure that messages are sent before their deadlines expire. To give such guarantees, an admission control, based on feasibility analysis, is required before admitting a new real-time stream to the network. So no streams are admitted that cause other streams to miss their deadlines.

*Non-real-time traffic:* besides periodic real-time traffic our network protocol should allow for best-effort traffic, which must not affect the deadlines of real-time streams. The only constraint is that the network reserves a certain percentage of the available bandwidth for non-real-time traffic.

*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. Note, however, that deadline misses are impossible to rule out when network faults occur.

*Plug-and-play:* when a new node is added to the network it should be recognized by the network protocol and automatically added to the schedule. When a node goes off-line the network should also remove the node from its internal data structures and update the schedule.

*Can be based on existing hardware and protocols:* it must be possible to use cheap existing hardware and software. This requires that the network used should at least be based on existing hardware. Customized hardware would drive up the price of experimental devices considerably.

## 3. Related work

Broadcast networks are typically set up as a shared-bus network, where any node can talk directly to all other nodes. Ethernet [1] [1] is the best-known example of such a network. Ethernet networks operate at a speed of 10 Mbit/s or 100 Mbit/s and can typically be found in home and office environments. Ethernet networks that operate at 1000 Mbit/s are beginning to be used in data and network centres.

Common-bus network protocols need a way to resolve collisions. Ethernet uses an exponential back-off algorithm to do this. This algorithm is not deterministic so, in order to build a real-time network on top of it, determinism will have to be enforced by using some algorithm to avoid collisions. There have been and there still are people working on creating deterministic Ethernet networks, enabling it to be used for real-time purposes [2].

The simplest approach to have some guarantees on a bus network is by using a TDMA (Time Division Multiple Access) approach. Providing guarantees on bandwidth can also be achieved by putting contraints on the traffic generated by the nodes. There are three classes of methods to do this, without making changes to the Ethernet protocols: traffic smoothing, virtual time protocols, and window protocols.

The traffic smoothing approach [3, 4] uses a statistical method to bound the medium access time by limiting the packet arrival rate at the Medium Access Control (MAC) layer. Thus non-real-time traffic bursts are smoothed: they are spread out over a longer time period, to allow real-time traffic to use the network as soon as it arrives.

Virtual time protocols, first proposed by Molle and Kleinrock [5], implement a packet release delay mechanism. Zhao and Ramamritham [6] describe several virtual time CSMA (Carrier Sense Multiple Access) protocols, in which the virtual clocks differ. Recently Salian et al. [7] improved on the minimum-deadline-first variant of Zhao and Ramamritham to support both hard real-time and soft real-time messages. El-Derini and El-Sakka [8] proposed a virtual time CSMA variant which uses two priority classes, where higher-priority messages will be given precedence when sending. Molle [9] improved the original protocol, allowing for a certain number of priority classes. His simulation results indicate the performance of this prioritized virtual time CSMA protocol compares very favourably to other prioritized CSMA protocols [10–12].

Gallager [13] was the first to propose a window protocol, in which the enabled set of stations are those that have messages in the queue which were generated during some interval, or window. The traditional window protocol, as proposed by Gallager, has been modified to allow implementation of Minimum Laxity First (MLF) or other policies, to make it suitable for real-time applications. Zhao et al. [14] based the window on the latest time to send a message to make it meet its deadline. Znati [15] built a simulation of a window protocol which implements the MLF strategy. Collisions, resulting from messages with equal laxity, are resolved using a MLF policy preserving contention-parameter selection, based on a node's latest time to send and this node's position in the logical ring, in which all nodes have been placed for this protocol.

The real-time network RETHER [16–18], developed at the State University of New York at Stony Brook, is based on Ethernet. It uses a token-based technique to regulate access of all the nodes to the network. This circumvents the danger of collisions inherent to the CSMA/CD (Carrier Sense Multiple Access with Collision Detection) protocol employed on Ethernet networks.

Many solutions exist to create a real-time network from an existing non-real-time, fully-connected, broadcast capable network. However, we consider them inadequate for our needs. TDMA is inflexible and needs a master node. This creates a single point of failure, when this node leaves the network. Traffic smoothing's statistical method by its nature cannot guarantee real-time demands 100% of the time. Virtual time and window protocols are complex, making it hard to implement them in resource-lean devices. RETHER inspired us to do this work. RETHER uses a long time to switch to real-time mode when the first real-time stream arrives and uses a less than optimal real-time scheduling strategy. Our approach does not have these shortcomings as we will show in this paper.

## 4. Design of the protocol

This section describes our real-time token network protocol. Its specification meets the design requirements given in section 2.

### 4.1. Overview

Our protocol provides real-time communication facilities in a distributed manner. It can be implemented on any network standard which provides a fully-connected network with broadcast capability. For building a prototype we selected Ethernet because it is readily available and cheap. The Ethernet hardware is not changed.

We use a token as a form of shared memory between the nodes. The main difference between our protocol and other token-based protocols is the way the token is distributed to the nodes in the course of time. E.g., RETHER uses a simple static round-robin method. Our network uses preemptive Earliest Deadline First (EDF) [19]. Any other desired real-time scheduling algorithm, such as Rate Monotonic (RM) [20] or Deadline Monotonic (DM) [21], can

[1] This paper will use the term Ethernet throughout for networks adhering to the IEEE 802.3 standard.

also be chosen. These scheduling algorithms distribute the token according to bandwidth demands. We also use on-line feasilibity analysis for admission control of new streams. It is assumed that all nodes in the network collaborate on this distributed real-time scheduling policy.

A fully-connected, single-segment network is required because any node needs to be able to forward the token to any other node directly. The broadcast capability of our network is used to handle efficient addition of new nodes.

Our protocol also provides a mechanism to deal with token loss, token duplication or deadline misses due to network faults. This is done by introducing the *monitor* state of a node. When a node is in the monitor state it keeps a copy of the token, and can send it to the target node when the token holder fails to do so. This prevents a single point of failure, since token loss requires two nodes to fail simultaneously: the token holder and the monitor. In case of faulty behaviour it cannot be guaranteed that all deadlines are met.

### 4.2. The token

The original idea of using a token comes from the timed-token MAC protocol [22, 23]. A large amount of research regarding the token protocol has been done and many real-time network protocols use some sort of token algorithm to provide QoS guarantees. The main advantage of using a token is collision prevention. This is achieved by allowing only one node, the token holder, to transmit messages across the network at any given time. Secondly, by letting each node hold the token for a pre-designated time and using a deterministic algorithm to select the next node that is eligible for the token, QoS guarantees can be given.

When a node holds a token, it may transmit messages across the network. This can be real-time or non-real-time traffic. The traffic type depends on the schedule. The time a node may transmit messages is called the Token-Holding Time (THT). Unlike the original timed-token MAC protocol, which uses a static, fixed THT for every node, the THT in our protocol is determined by the network schedule.

The information kept in the token is the list of participating nodes, their addresses, and the list of currently active streams. Per stream the token contains its type, its source node, its bandwidth, its period, its remaining transmit time for the current period, and its ready time, i.e. the next time it will start a new period.

When the THT has expired the network scheduler determines the next node that may transmit and calculates its THT. This is possible, because all network state that is required for this calculation is kept in the token. Next the token holder goes into the monitor state and transfers the token to the node with the highest priority, while the previous node in monitor state will finish its monitor function.

The role of the monitor is to generate a new token in case the current token is lost, either because the token message is lost or the token holder dies. We assume that the monitor and the token holder do not die simultaneously.

This process of sending messages and forwarding the token continues forever. The token holder is primarily responsible for forwarding the token and the monitor is its backup. Both the token holder and monitor role circulate among the nodes. The token holder and monitor role may be fulfilled by one and the same node only if there does not exist a previous token holder. This is the case when a fault has been detected and the monitor decides it needs the token first, when a network is being started or when a network consists of one node.

### 4.3. The network scheduler

We use a real-time scheduling strategy to determine which node may get the token and thus may transmit messages across the network. There is a wide range of schedulers available for managing processes on a CPU. Most schedulers can also be used to schedule traffic on a real-time network. Our current prototype uses pre-emptive EDF, which has a simple scheduling algorithm and an even simpler feasibility analysis, while still being able to utilize the network up to 100%. An alternative would be non-pre-emptive or mixed-pre-emptive EDF scheduling, which would enable the use of passive nodes on the network. However, this needs a more complex admission check, which increases the overhead from the protocol [24]. Using RM or DM instead of EDF is another alternative, these schedulers are more predictable if occasional overload situations are tolerable.

In contrast to an EDF scheduler which deals with processes on a CPU, our scheduler is replicated on all nodes on the network. Every node has to be able to calculate a schedule and to update the schedule when streams are added or removed. So all scheduling information has to travel along the network as well. We use the token as a container for all necessary scheduling information to enable all nodes to make the correct scheduling decisions, like to which node the token is transferred when the current node is finished transmitting.

To use this kind of scheduling reliably on a network a few provisions are needed. The maximum time for a packet to travel across the network from node $x$ to node $y$ needs to be fixed and known. Also all nodes need to have the same notion of time: their clocks need to be synchronized (see section 4.6).

A network contains a set of nodes, each of which may send real-time traffic. The real-time traffic can be split into separate streams. Each stream is a one-way channel between two nodes through which data can be sent. A node may use more than one stream.

At any given time at most one real-time stream can be active, thus other streams will have to wait. The scheduler

decides which stream may transmit and when. Basically a stream that wants to transmit on the network is analogous to a task that wants to execute on a CPU in a multitasking system. In this way known scheduling algorithms, like First-In First-Out (FIFO), RM, DM, and EDF can be applied when used on a network. It is possible to apply techniques such as pre-emption, non-pre-emption, blocking, and feasibility analysis in the same way.

Schedule parameters of tasks and streams differ as follows:

*Bandwidth:* the bandwidth parameter replaces the computation time parameter of a task. The required bandwidth of a stream is the minimum number of bytes or octets that the network should be able to transmit in a time of 1 second.

*Period:* the period of a stream is the time difference between two successive messages. This means that when each period has elapsed a new message is ready to be transmitted over the network. Also, the previous message should be received completely before the period expires. In other words, the deadline of a message is the same as its period.

**4.3.1. Feasibility analysis.** Whenever a new stream is introduced to the network, a feasibility analysis should be performed. The same parameters for task scheduling, like execution time and period, used in these feasibility analyses can be retrieved from the network streams. An interesting parameter is the utilization $U_i$ of a stream $i$:

$$U_i = \frac{B_i}{B} = \frac{C_i}{T_i} \tag{1}$$

where $B_i$ is the network bandwidth of stream $i$ and $B$ is the maximum bandwidth available on the network. We consider a stream on the network analogous to a task executing on a processor using a worst-case execution time $C_i$, a period of execution $T_i$, and a relative deadline $D_i = T_i$.

For pre-emptive EDF with $D_i = T_i$ the following equation gives a necessary and sufficient way of determining the feasibility of a set of tasks [19]:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2}$$

Combining equations 1 and 2 yields the following equation to determine the feasibility of a set of real-time streams on our network, using pre-emptive EDF as the scheduler:

$$\sum_{i=1}^{n} \frac{B_i}{B} \leq 1 \tag{3}$$

So, to check if adding a new stream is feasible, it is only necessary to calculate the total utilization of all streams, including the new stream. If this total utilization is lower than or equal to 1, the system remains feasible and the new stream can be added. Otherwise, the new stream is rejected, and the application requesting the new stream may try again using lower QoS parameters.

Because our network also can reserve bandwidth for non-real-time traffic, the maximum utilization of all real-time streams is set to a value lower than 100%. The optimal maximum real-time utilization depends on the application area and likely varies with time. An appropriate fixed maximum will have to be determined experimentally for each application area.

Streams may pre-empt each other. When stream $i$ pre-empts stream $j$ this means that stream $i$ has a later release time and an earlier deadline than stream $j$. The node that schedules stream $j$ already knows when the pre-emption will occur. This might not be called pre-emption per se, but normal real-time pre-emptive scheduling theory is still valid. It computes a THT that will allow stream $j$ to use the network until stream $i$ is released. The node from which stream $j$ originates will then compute a THT for stream $i$ and forward the token to the source of stream $i$. When stream $i$ is finished, the token will return to the source of stream $j$, provided there is no other stream $k$ that would also pre-empt stream $j$.

This computation of how long a stream may make use of the network makes it possible for the upper network layers to adapt the packet size used. To increase network efficiency larger packet sizes may be used when a stream has a longer consecutive amount of time than when a shorter amount is available.

Streams are assumed to be independent and pre-emptable. It is possible to use non-pre-emptable streams. This has the advantage that the scheduling algorithm is simplified, as it can just use the network for the allocated time, without having to take other streams into account. However, non-pre-emptability puts additional burden in the admission control part. To calculate the feasibility of a set of non-pre-emptable streams, blocking between streams has to be taken into account. Selection of pre-emptable or non-pre-emptable streams depends on the application area: a trade-off has to be made and the application will determine the better choice.

**4.3.2. Token and network overhead.** In the feasibility analysis in equation 3 only the bandwidth of the real-time stream is considered. Sending the token costs network bandwidth, as well as the overhead the physical layer puts on each packet. To account for this overhead, the utilization calculation of a real-time stream has to be refined. We assume the computation time of this calculation is negligible compared to the transmission times, and do not take it into account. Without loss of generality we choose for a collision-free Ethernet network as an example.

First we account for overhead. Assuming maximum-sized Ethernet packets are used for real-time packets the actual bandwidth used by a real-time stream can be calculated as follows:

$$B'_i = B_i \frac{1538}{1500} \tag{4}$$

where 1538 is the size of an Ethernet packet including the 18 octet header and 20 octet interframe gap,[2] and 1500 is the size of the actual payload of an Ethernet packet.

This corrected bandwidth $B'_i$ is used as the actual stream bandwidth for the scheduling algorithm and feasibility analysis.

Every time a stream becomes ready it "requests" the token. Thus before a stream can start transmitting, time is needed to send the token from the previous node holding the token to the node "requesting" the token. When a stream has finished transmitting a message the token has to be forwarded to the next node. This token transmission can be accounted for in the stream running from that node.

When a stream pre-empts another stream, an additional token transmission is needed, because the token has to be routed back to the node which was pre-empted. This accounts for one additional token transmission per pre-emption. So, in the worst case, where every stream invocation pre-empts another stream invocation, every message has an overhead of sending the token twice. In case of non-pre-emptive message transfers this is *only once*.

When a token is sent to the next host, it is actually broadcast. This serves as a confirmation message for the *monitoring* node. Altogether this makes the token bandwidth overhead equal to twice the token size, divided by the period of the real-time stream. The corrected bandwidth for a stream can thus be calculated using the following formula:

$$B_i^{Corrected} = B'_i + 2B_{token} = B'_i + 2\frac{S}{T_i} \tag{5}$$

where $B_{token}$ is the token bandwidth overhead and $S$ is the size of the token packet, including physical layer headers.

The feasibility analysis algorithm should use this corrected bandwidth to compute the feasibility of a set of real-time network streams. The total overhead depends on packet size and stream periods.

**4.3.3. Deadline misses.** A deadline miss will occur when a node receives the token late, because of some network fault, and consequently cannot complete its transmission before its deadline. When a node detects this, it does not transmit or finish transmitting the current message, but it informs the sending application what happened and passes the token to the next node, so the token at least arrives there on time. The application may be sent an exception with details regarding which part of a real-time stream has not been sent correctly. The application owning the real-time stream can then take appropriate action to account for the message that went wrong.

[2]The interframe gap does not contain 20 actual octets, but it takes as much time as is needed to transmit 20 octets.

**4.3.4. Non-real-time traffic.** The network permits non-real-time traffic when no real-time streams want to transmit something. The token goes around the network in a round-robin fashion to permit all nodes to send non-real-time messages. As soon as a real-time stream becomes ready, the token should go to the corresponding node, so real-time traffic can commence. The last node visited for non-real-time traffic should be remembered, so the next round of non-real-time messages can restart at that node.

## 4.4. The operation of the protocol

Every node in the system executes a state machine. This state machine is shown in figure 1. Most error-induced transitions are left out to keep the state machine comprehensible.

First we describe the network as it operates normally, this includes the function in the *Idle* state, the *Activate* state, the *Dispatch* state, the *Transmit* state, and the *Monitor* state. Then we will describe the on-line addition and removal of nodes, which includes the function in the *Offline* state and the *Announce* state. This is followed by a description of stream addition and removal. We will describe how the protocol handles faults, including the function of the *Poll* state, and finally we give a short description of the clock synchronization needed by the protocol.

**4.4.1. Normal operation.** The protocol works in rounds. Whenever the token travels to a new node, a new round is started.

At the receipt of a token in the *Idle* state, the node moves to the *Activate* state (transition *A* in figure 1). One of its streams is selected for transmission. The node checks the active stream in the token. If the stream is valid, the node will start transmitting data (transition *B*) and enter the *Transmit* state. If the stream is invalid, because the application terminated the stream, the node will not begin transmission, but computes a new schedule immediately (transition *C* in the figure).

The most important, and central, state is the *Dispatch* state. In the *Dispatch* state the decision is made which stream is next to access the network. The scheduler thus executes this state. Depending on the stream that is next, several things may happen. The protocol is assumed to spend a negligible amount of time in the *Dispatch* state.

If the next stream originates from another node, the token has to be forwarded to that node. The THT for only the next stream is computed and put inside the token. The token is broadcast to the other node, which becomes the token

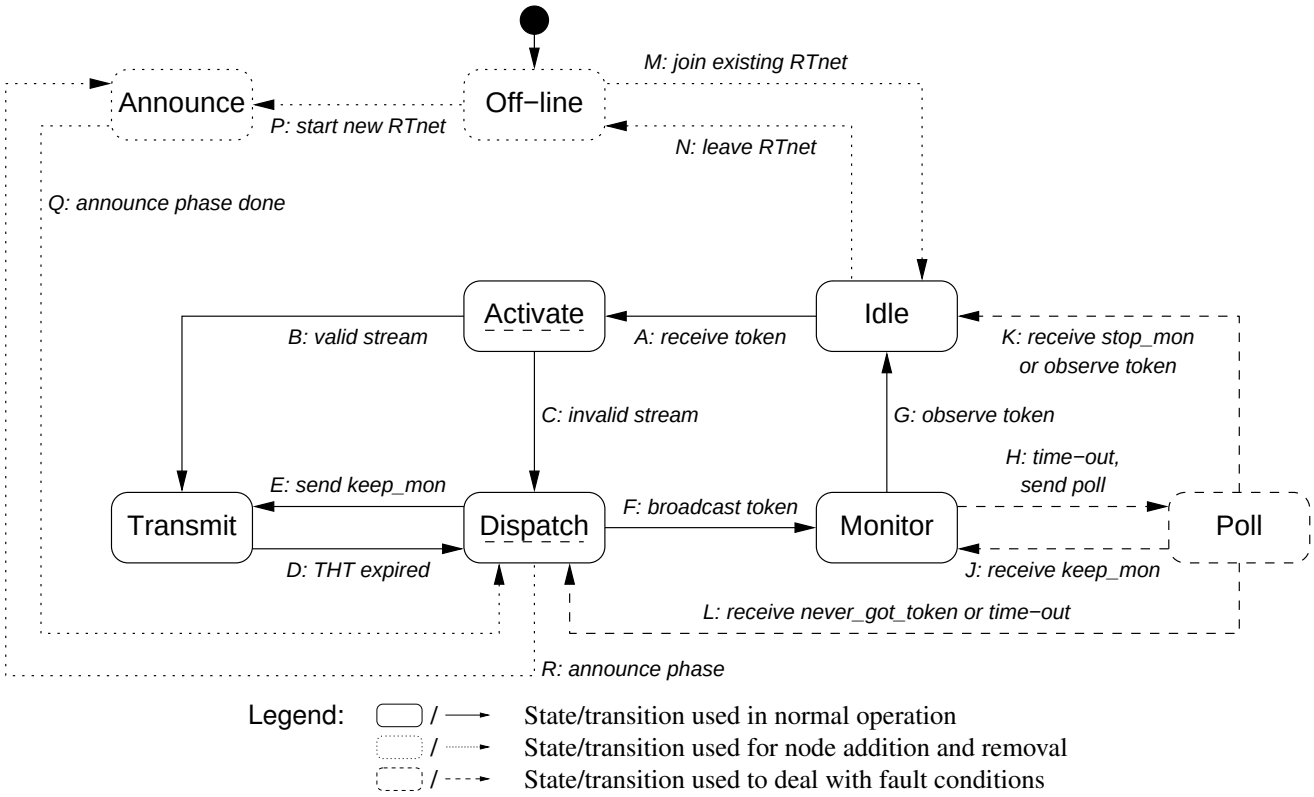


Figure 1. RTnet state-transition diagram

holder. Simultaneously the monitor notices the progress of the token and stops monitoring. Then the node moves into the *Monitor* state, and becomes the monitor itself (transition *F* in the figure).

If the next stream originates from the current node, the token remains in that node and is not forwarded. The monitoring node is told to *keep monitoring* for the new THT, as it was only monitoring for the THT just elapsed. The node moves into the *Transmit* state (transition *E*), where it stays as long as needed to transmit the amount of data for the current stream. When the THT expires or there is no more data to be sent for the current period, the node enters the *Dispatch* state (transition *D*).

If the next stream is the announcement or synchronization stream, a special phase is entered. These are described respectively in sections 4.4.2 and 4.6.

The monitoring node moves into the *Idle* state when it observes the token being forwarded elsewhere (transition *G*). When the token is sent to the monitoring node, it will immediately leave the *Idle* state again and proceed to the *Activate* state (transition *A*).

Streams may only be added or removed when a node has the token. An admission check is carried out before a stream may be added. The network is configured not to allow more than, say, 90% of real-time traffic, to guarantee there is some bandwidth left for non-real-time data and to provide slack for activities that are assumed to take a negligible amount of time. To guarantee that every node regularly receives the token, a special stream, the *token receive* stream, is allocated for every node with a period of 3 s and a bandwidth of one physical layer frame per period.

**4.4.2. Node addition and removal.** At initialization time every node starts in the *Off-line* state. It then waits for an amount of time equal to two times the announcement period $T_{announce}$ (= 2 s). If during that time a special invitation message, asking for new participants in an already running RTnet, is received, the node replies that it would like to join that RTnet. It then moves into the *Idle* state (transition *M*).

When no invitation is received, apparently RTnet is currently inactive. The initializing node, whose invitation timeout will expire first, creates a new token, containing only itself, and two streams. These are its own *token receive* stream and the *announcement* stream. The node then broadcasts an invitation for others to join, and moves into the *Announce* state (transition *P*). It waits a short time (10 ms) for replies, adding replying nodes to the token, with a *token receive* stream for each of them. Next it enters the *Dispatch* state, in which it will be decided to which node the token will be transferred (transition *Q*).

On an already running network, a special stream is present for handling the addition of new nodes: the *an-*

*nouncement* stream. When this announcement stream becomes active, the node broadcasts an invitation message and moves into the *Announce* state (transition *R*). It then again waits a short time (10 ms) for replies, adding replying nodes to the token, with a *token receive* stream for each of them. Next it enters the *Dispatch* state (transition *Q*) and proceeds as described above.

The announcement stream is currently bound to the node that first started the network. When this node dies, the monitor will notice this and remove its streams from the token. It will detect that it had the *announcement* stream and the monitor will take over this stream.

A node may only remove itself when it has the token. It needs to remove its own streams from the token, remove itself from the token, and forward the token to the next node. But it has to perform monitor duties for one more round before it may actually disconnect. Only then the node may move from the *Monitor* state to the *Off-line* state (transition *N*).

**4.4.3. Stream addition and removal.** A node may only add streams when it is the token holder, i.e. when it is in the *Dispatch* state or the *Transmit* state. To ensure schedulability an admission check has to be carried out before a stream may be admitted, as described in section 4.3.1.

Removing streams can never cause an infeasible schedule of streams, therefore it is not necessary to perform a schedulability analysis.

## 4.5. Fault handling

The fault handling of the protocol should cover the most important faults. It should also not burden the normal behaviour, and it should be light-weight, to enable implementation in resource-lean devices.

The network is able to recover from the most important faults: loss of a token or any other single protocol control packet and failure of a single node. It cannot recover when the token holder and monitor fail simultaneously or when two or more consecutive protocol-related packets are lost. The fault situations and the protocol's handling of them covered by the current design will be described next.

When faults occur and need to be handled, we do not give real-time guarantees any more until the fault is resolved. In this time it could occur that more than one message are put on the network at once. Usually, however, in such a case all messages that are put on the medium at the same time are lost and this should be prevented.

Faults are detected by the monitor. When the monitor does not perceive the token being broadcast in time, determined by $THT + \delta_{monitor}$, it will have to find out if the token really has been lost. This is done by polling the token holder. When the token has not been lost, and some node is actively using the network, it is not necessary to send a *poll* message.

If the monitor detects activity on the medium from another node, it may conclude the token was forwarded correctly and it just missed this fact. The monitor can then stop monitoring, since the node it was monitoring has become monitor for the new token holder.

If the monitor detects activity from the node it was monitoring, it may conclude it missed a *keep monitoring* message. Now it will have to actively monitor the token holder by observing its traffic, since the monitor does not know the current THT. It can resume normal behaviour as soon as a new *keep monitoring* message or a new token is observed.

When no network activity is detected, the token was lost. Either the token transmission failed, or the token holder itself died. To distinguish between these possibilities, the monitor will send a *poll* message to the token holder and move into the *Poll* state (transition *H*).

When the token holder is still alive, it will respond to the poll with a message to the monitor. There are three possible responses. The first is that the token holder forwarded the token and is now monitoring. Then the erroneous monitor will be told to stop monitoring (transition *K*). It is possible that this token was not forwarded correctly, but it is now the new monitor's responsibility to resolve that situation.

Secondly the token holder can reply that it never received a token. The monitor then moves into the *Dispatch* state (transition *L*), to determine the stream which may now use the network. This is not necessarily the same stream as computed earlier, but is determined by the scheduling algorithm and the policy about missed deadlines.

Finally it is possible that the token holder is no longer alive, the monitor will notice this after a timeout $\delta_{monitor}$(= 50 ms). It will then enter the *Dispatch* state (transition *L*). It will remove the, now dead, token holder from the token, along with any streams originating from that node. Then the next stream will be selected for transmission and the protocol will continue operating normally.

Using detection of network activity requires the token holder to be not silent. When the token holder does not have actual data to send, it should periodically transmit a packet signifying it is still alive. This bounds the time a monitor will have to listen to detect activity on the medium.

When a token is received by a node that is not in either the *Idle* state, the *Monitor* state, or the *Poll* state, something is wrong: there are two tokens in the network. This can happen when more than one protocol-related control messages are lost in a short amount of time. Both the token holder and the monitor then forward the token. When the token holder has not added or removed some streams, the scheduler in both nodes will select the same stream, and thus the same node to forward the token to. This node will then receive two tokens. This error condition is not shown in figure 1. In this case the monitor mentioned in this extraneous token is told to stop monitoring, essentially killing all state present in that token.

When the physical layer is capable of collision detection, it is possible to simplify the protocol somewhat. It is then not necessary to listen before transmitting a poll, it can be done immediately when the monitor experiences a time-out. This poll can interfere with actual data on the network, but the physical layer's collision mechanism will resolve this automatically, at the cost of deterministic transmissions. The polling monitor can then receive a new *keep monitoring* message to replace a lost one, when the token holder still has a valid token (transition *J*).

Fault handling is a complex issue. We have an analysis using the Uppaal software tool [25]. It is beyond the scope of this paper to handle all the details here.

### 4.6. Clock synchronization

As our protocol uses a distributed scheduling algorithm, where each node may take scheduling decisions, it is necessary that every node has the same view on time in the system. The synchronization algorithm will not be presented in detail, as it is outside the scope of this paper, only the highlights will be given.

Synchronization is based on the principle of reference broadcasts [26]. All nodes, including the synchronization master, measure the delay between two broadcasts with their local clock and inform the synchronization master. This node will compute a common clock rate and distribute this to the others using a third broadcast and these modify their local clock rate accordingly.

Currently the scheduler uses a granularity of 10 ms. The synchronization algorithm maintains less than 2 ms distance between the clocks. This is sufficiently accurate with respect to the scheduler granularity and the nodes' clocks are running close enough for the scheduling algorithm to make the same decisions on every node.

## 5. Performance

The performance tests are carried out on a prototype of the protocol, implemented in the Linux operating system on standard Ethernet hardware. A network of 7 nodes is used, connected with two 10 Mbit/s networks. All nodes are ordinary PCs, ranging from a Pentium Pro based PC to a Pentium 3 based one. All nodes are running a standard Linux kernel.

One network is used for running our protocol, the other is used for communication between the test software on the nodes and the test control software running on an eighth node, that doubles as measurement station. This eighth node participates passively in the time synchronization protocol of RTnet to enable time-accurate measurements.

The tests are run on four different network configurations. Two parameters are varied: the presence and absence of non-real-time traffic and the use of more and less accurate clock sources. The clock synchronization algorithm is able to use the Pentium TSC cycle counter which is more accurate than the standard 8254-based clock of a PC. As we feel the protocol should be able to run on less-powerful processors than a Pentium, we test with the less accurate PC clock as well, as small devices will generally only have such a simple, crystal-driven clock.

### 5.1. Net bandwidth

We execute two test sets: a set of one or two simple, regular streams between two to four nodes to get an indication of the raw performance and two complex scenarios with varying network load, up to 90%. The complex scenarios have one large-bandwidth, large-period stream and many small-bandwidth, small-to-large-period streams. This is to increase the amount of pre-emptions of the large stream. All streams are based on UDP/IP with two datagram sizes: 5912 and 8192 B. The first fits exactly in four Ethernet frames, the latter is the default size in Linux. Table 1 lists the average net bandwidth obtained for the tested streams. In all cases, the average bandwidth is close to the requested bandwidth, from which we may conclude that the protocol works well.

Despite the fact that the differences are small we have analysed the causes and identified two possibilities. The first cause stems from the use of UDP. A UDP datagram is split into several Ethernet frames, of which one frame is not always of maximum size. This results in a somewhat less efficient use of the medium. The second cause is due to the lack of real-time scheduling precision of our Linux kernel and would not have occurred if we had used a proper real-time kernel.

### 5.2. Overhead

According to equation 5 the overhead is low for streams with larger periods and it increases for shorted periods. Figure 2 depicts the worst-case gross bandwidth, i.e. bandwidth including overhead from tokens and control messages, for four bandwidths and a range of periods. It also shows the average measured gross bandwidths for some combinations of bandwidth and period. For shorter periods the actual gross bandwidth is less than worst-case. The size of the token grows with the number of streams, and in our tests the tokens were maximally 1002 B.

### 5.3. Initial stream latency

Between the time an application requests a real-time stream and the time its request is granted or rejected, the node will wait for a token, compute the feasilibity of adding the stream, and then inform the application about the stream. An application therefore experiences a latency before it may start using the stream.

Table 1. Net bandwidth of tested streams

| requested BW (kB/s) | period (s) | # of tests | average BW (kB/s) | BW std. dev. |
|---|---|---|---|---|
| 0.5 | 3.1 | 100 | 0.497 | 0.001 |
| 0.6 | 2.7 | 40 | 0.597 | 0.000 |
| 0.6 | 2.9 | 40 | 0.597 | 0.000 |
| 1.0 | 1.7 | 100 | 0.998 | 0.001 |
| 1.0 | 1.9 | 80 | 0.998 | 0.001 |
| 1.0 | 2.3 | 40 | 0.999 | 0.000 |
| 1.0 | 3.1 | 60 | 0.998 | 0.001 |
| 1.1 | 2.9 | 40 | 1.098 | 0.001 |
| 1.2 | 2.7 | 60 | 1.198 | 0.001 |
| 1.4 | 2.3 | 40 | 1.398 | 0.001 |
| 1.6 | 1.9 | 60 | 1.597 | 0.000 |
| 1.8 | 1.7 | 120 | 1.798 | 0.001 |
| 2.0 | 0.7 | 20 | 2.000 | 0.000 |
| 2.0 | 1.1 | 60 | 1.998 | 0.001 |
| 2.0 | 1.3 | 80 | 1.998 | 0.001 |
| 2.4 | 1.3 | 80 | 2.398 | 0.001 |
| 2.8 | 1.1 | 40 | 2.798 | 0.000 |
| 4.0 | 0.3 | 100 | 3.999 | 0.001 |
| 4.0 | 0.5 | 80 | 3.998 | 0.000 |
| 4.4 | 0.7 | 60 | 4.398 | 0.000 |
| 6.1 | 0.5 | 80 | 6.098 | 0.001 |
| 8.0 | 0.2 | 20 | 7.997 | 0.000 |
| 10.1 | 0.3 | 60 | 10.097 | 0.001 |
| 15.2 | 0.2 | 40 | 15.198 | 0.001 |
| 100.0 | 0.05 | 360 | 99.986 | 0.018 |
| 100.0 | 0.1 | 680 | 99.995 | 0.005 |
| 100.0 | 0.5 | 680 | 99.990 | 0.013 |
| 100.0 | 1.0 | 680 | 99.994 | 0.008 |
| 100.0 | 2.0 | 680 | 99.990 | 0.009 |
| 100.0 | 5.0 | 680 | 99.989 | 0.011 |
| 200.0 | 0.05 | 680 | 199.994 | 0.002 |
| 200.0 | 0.1 | 680 | 199.990 | 0.006 |
| 200.0 | 0.5 | 680 | 199.994 | 0.002 |
| 200.0 | 1.0 | 680 | 199.990 | 0.010 |
| 200.0 | 2.0 | 680 | 199.993 | 0.008 |
| 200.0 | 5.0 | 680 | 199.993 | 0.009 |
| 500.0 | 0.5 | 2600 | 499.987 | 0.011 |
| 500.0 | 1.0 | 2600 | 499.984 | 0.008 |
| 500.0 | 2.0 | 2760 | 499.987 | 0.005 |
| 500.0 | 5.0 | 2760 | 499.986 | 0.005 |
| 500.0 | 30.0 | 40 | 500.000 | 0.000 |
| 1000.0 | 0.5 | 40 | 999.986 | 0.003 |
| 1000.0 | 1.0 | 40 | 999.990 | 0.001 |
| 1000.0 | 2.0 | 40 | 999.986 | 0.003 |
| 1000.0 | 5.0 | 40 | 999.985 | 0.004 |

The frequency of *initial stream latencies* that occurred during all our tests are shown in figure 3. The minimum latency observed was 269 µs and the maximum latency observed was 3.79 s. Note that the initial stream latency may be anywhere between 0 s and two times the period of the token receive stream (2 times 3 s).

Figure 3 shows the amount of latencies that occurred in bands with a width of 100 ms. The lowest band has been split into two bands from 0 ms to 1 ms and from 1 ms to 100 ms, because 15% of all latencies is less than 1 ms. Figure 3 clearly shows the distribution of latencies: 53% of the latencies is less than 500 ms and 76% of the latencies is less than 1 s. The latencies are very reasonable for applications. If smaller initial stream latencies are needed, the period of the token receive stream may be lowered at the cost more token transmissions and thus overhead.

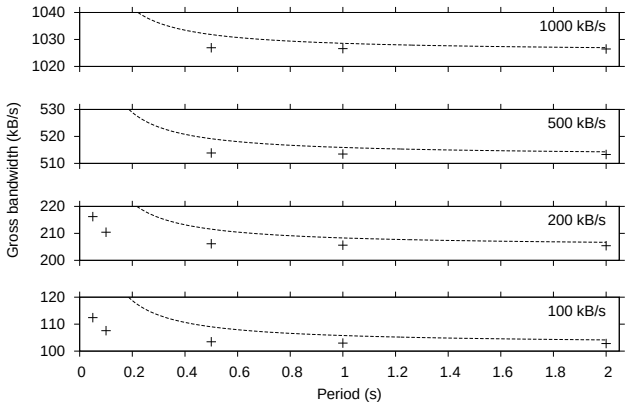


Figure 2. Worst-case gross bandwidth usage of RTnet protocol (lines) with measurements of actual gross bandwidth (points)

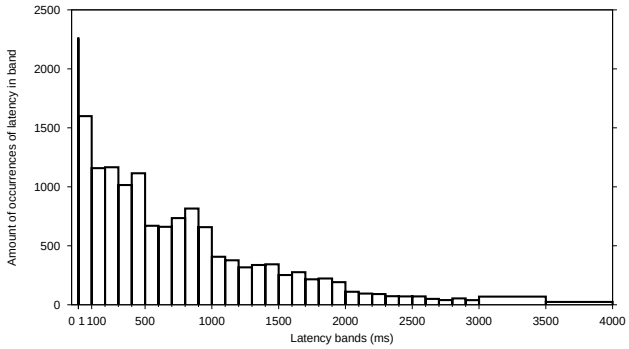


Figure 3. Occurrences of initial stream latency lengths

## 6. Future work

We will investigate the performance of our protocol on 100 Mbit/s Ethernet and compare this performance to the use of normal Ethernet on an ordinary, cheap, run-of-the-mill 100 Mbit/s Ethernet switch.

Furthermore we will look into other physical networks on which to run the protocol. Mainly we will look into the possibility of putting RTnet on top of currently available or future wireless ad-hoc networks, such as Wavelan, Bluetooth, or new experimental networks.

## 7. Conclusions

This paper has shown the design of the distributed real-time protocol RTnet that enables a broadcast-capable network to provide real-time stream capabilities. The RTnet

protocol's design adheres to the following requirements: it provides QoS guarantees, it allows non-real-time traffic, is tolerant to a certain degree of packet loss and node failure, it allows for random node addition and removal, and it can run on cheap, existing hardware, such as Ethernet. Subsequently the protocol's operation was described, and some results from a prototype implementation were presented.

The distributed RTnet protocol behaves fully according to expectations for streams with larger periods. But, as expected, it shows some overhead for smaller periods. It may be used with pre-emptive or non-pre-emptive EDF as a scheduling paradigm. RTnet can also make use of RM, DM, or any other real-time scheduler.

## Acknowledgements

We would like to thank Tjalling Hattink for his contributions in co-developing the ideas and creating an initial simulation environment. Also we would like to thank Robert Krikke for his extensive checks of the state-transition table, Jean-Paul Panis and Theo van Klaveren for their help in debugging the prototype, and Pieter Hartel and Vasughi Sundramoorthy for their valuable comments on the paper.

## References

[1] *IEEE Standard for Local and Metropolitan Area Networks — Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, IEEE, 2002, IEEE Std. 802.3-2002.

[2] M. Alves, E. Tovar, and F. Vasques, "Ethernet goes real-time: a survey on research and technological developments," Polytechnic Institute of Porto — School of Engineering (ISEP-IPP), Tech. Rep. HURRAY-TR-0001, Jan. 2000.

[3] S.-K. Kweon, K. G. Shin, and Q. Zheng, "Statistical real-time communication over Ethernet for manufacturing automation systems," in *Proc. 5th IEEE Real-Time Technology and Applications Symposium*, June 1999, pp. 192–202.

[4] S.-K. Kweon, K. G. Shin, and G. Workman, "Achieving real-time communication over Ethernet with adaptive traffic smoothing," in *Proc. 6th IEEE Real-Time Technology and Applications Symposium*, May/June 2000, pp. 90–100.

[5] M. L. Molle and L. Kleinrock, "Virtual time CSMA: why two clocks are better than one," *IEEE Trans. on Comm.*, vol. 33, no. 9, pp. 919–933, 1985.

[6] W. Zhao and K. Ramamritham, "Virtual time CSMA protocols for hard real-time communication," *IEEE Trans. on Softw. Eng.*, vol. 13, no. 8, pp. 938–952, 1987.

[7] S. F. Salian, A. Y. M. Shakaff, and R. B. Ahmad, "Improvement of virtual-time CSMA protocol for distributed hard and soft real-time systems on the Ethernet," in *Proc. 2002 Stud. Conf. on Research and Development*, July 2002, pp. 128–131.

[8] M. N. El-Derini and M. R. El-Sakka, "A CSMA protocol under a priority time constrained for real-time communication," in *Proc. 2nd IEEE Workshop on Future Trends of Distributed Computing Systems*, Sept./Oct. 1990, pp. 128–134.

[9] M. L. Molle, "Prioritized-virtual-time CSMA: head-of-the-line priority classes without added overhead," *IEEE Trans. on Comm.*, vol. 39, no. 6, pp. 915–927, 1991.

[10] F. A. Tobagi, "Carrier sense multiple access with message-based priority functions," *IEEE Trans. on Comm.*, vol. 30, no. 1, pp. 185–200, 1982.

[11] I. G. Niemegeers and C. A. Vissers, "TWENTENET: a LAN with message priorities, design and performance considerations," in *Proc. of the ACM SIGCOMM symp. on communications architectures and protocols*, 1984, pp. 178–185.

[12] C.-T. Lea and J. S. Meditch, "A channel access protocol for integrated voice/data applications," *IEEE J. on Sel. Areas in Comm.*, vol. 5, no. 6, pp. 939–947, 1987.

[13] R. G. Gallager, "Conflict resolution in random access broadcast networks," in *Proc. of the AFOSR Workshop in Communication Theory and Applications*, 1978, pp. 74–76.

[14] W. Zhao, J. A. Stankovic, and K. Ramamritham, "A window protocol for transmission of time-constrained messages," *IEEE Trans. on Comp.*, vol. 39, no. 9, pp. 1186–1203, 1990.

[15] T. Znati, "A deadline-driven window protocol for transmission of hard real-time traffic," in *Proc. 10th annual IEEE Conf. on Computers and Communications*, Mar. 1991, pp. 667–673.

[16] C. Venkatramani and T. Chiueh, "Supporting real-time traffic on Ethernet," in *Proc. 15th IEEE Real-Time Systems Symposium*, Dec. 1994, pp. 282–286.

[17] C. Venkatramani and T. Chiueh, "Design, implementation and evaluation of a software based real-time Ethernet protocol," in *Proc. of the conf. on Applications, technologies, architectures, and protocols for computer communication (ACM SIGCOMM '95)*, Aug./Sept. 1995, pp. 27–37.

[18] C. Venkatramani, "The design, implementation and evaluation of RETHER: a real-time Ethernet protocol," Ph.D. dissertation, State University of New York at Stony Brook, Jan. 1997.

[19] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[20] L. Sha, R. Rajkumar, and S. S. Sathaye, "Generalized rate-monotonic scheduling theory: A framework for developing real-time systems," *Proc. of the IEEE*, vol. 82, no. 1, pp. 68–82, 1994.

[21] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Hard real-time scheduling: the deadline-monotonic approach," in *Real Time Programming — Proc. Joint 17th IFAC Workshop on Real Time Programming and 8th IEEE Workshop on Real Time Operating Systems and Software*, May 1991, pp. 127–132.

[22] R. M. Grow, "A timed-token protocol for local area networks," in *Proc. Electro '82, Token Access Protocols*, May 1982, pp. 17/3:1–7.

[23] J. M. Ulm, "A timed token ring local area network and its performance characteristics," in *Proc. 7th IEEE conf. on Local Computer Networks*, 1982, pp. 50–56.

[24] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," in *Proc. 12th IEEE Real-Time Systems Symposium*, Dec. 1991, pp. 129–139.

[25] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *Int. J. on Softw. Tools for Techn. Transfer*, vol. 1, no. 1–2, pp. 134–152, 1997.

[26] J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," in *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Dec. 2002, pp. 147–163.