Design and implementation of a Real-time protocol over Ethernet

Iñaki Lazarobaster Badiola,
Igor Radovanović, Giovanni Russello
and Michel Chaudron

09/02/2004 - 17/09/2004
# TABLE OF CONTENTS

TABLE OF CONTENTS ........................................................................................................ 1  
ACKNOWLEDGMENTS ...................................................................................................... 2 
1. INTRODUCTION ............................................................................................................... 3 
2. CONTEXT .......................................................................................................................... 4 
   2.1. GSpace ........................................................................................................................... 4 
   2.2. RGSpace ....................................................................................................................... 5 
   2.3. Real-Time systems ......................................................................................................... 6 
3. REQUIREMENTS ............................................................................................................... 7 
4. ANALYSIS ........................................................................................................................ 10 
   4.1. Requirements analysis..................................................................................................... 10 
   4.2. Summary of existing real-time protocols over Ethernet .................................................. 11 
       4.2.1. RTPS: Real-Time Publish-Subscribe ........................................................................ 11 
       4.2.2. RTCAST: Lightweight Multicast for Real-Time Process Groups ................................ 14 
       4.2.3. Real-Time Ethernet Network at Home ...................................................................... 20 
       4.2.4. RT-EP: Real-Time Ethernet Protocol ........................................................................ 22 
       4.2.5. RETHER ................................................................................................................. 26 
       4.2.6. Adaptive Traffic Soothing ....................................................................................... 29 
   4.3. Comparison of existing protocols and choice .................................................................. 30 
5. DESIGNING TRIP ............................................................................................................. 32 
   5.1. Architecture .................................................................................................................. 33 
       Message ordering ............................................................................................................... 36 
   5.2. Message types .............................................................................................................. 36 
   5.3. Logical token ring algorithm ....................................................................................... 38 
   5.4. Class diagram .............................................................................................................. 39 
   5.5. Formulas ...................................................................................................................... 40 
       5.5.1. Definitions................................................................................................................ 40 
       5.5.2. Real-Time guarantees ............................................................................................ 42 
6. IMPLEMENTATION AND RESULTS .............................................................................. 46 
   6.1. Operating System .......................................................................................................... 46 
   6.2. Compiler ....................................................................................................................... 47 
   6.3. Implementation ............................................................................................................. 47 
   6.4. Results .......................................................................................................................... 47 
       6.4.1. Testing context ......................................................................................................... 48 
       6.4.2. Tests ....................................................................................................................... 48 
       6.4.3. Trip integration into RGSpace ............................................................................... 55 
7. CONCLUSIONS ................................................................................................................ 56 
8. REFERENCES ..................................................................................................................... 57
ACKNOWLEDGMENTS

There are several people who I would like to recognize for their help and support during my project. I would like to thank:

- Rob Spoor and Richard Verhoeven for their technical advices.
- Yarema Mazuryk and Thang Tran for their time and kindness.
1. INTRODUCTION

The **Design and implementation of a Real-time protocol over Ethernet** project was carried out in Eindhoven University of Technology, between February and September 2004.

In the Department of Mathematics and Computer Science, the System Architecture & Networking group is implementing a shared data space system (GSpace) and for its real-time extension (RGSpace) a real-time network protocol is needed. Design and implementation of such a protocol is the goal of this project.

The most convenient approach to design is to rely on existing standards and protocols that are widely accepted. This is the reason we have decided to use Ethernet protocol and the of-the-shelf hardware supporting it. Ethernet is simple to implement, low cost and most widely used protocol in Local Area Networks (LANs) nowadays and will most certainly be so for the time to come. The reason is that it is compatible with previous versions and upgrading it such that speed increases 10 times requires doubling the costs only.

When using Ethernet protocol, multiple end nodes might transmit simultaneously leading to a packet collision. Ethernet employs a shared media access control mechanism called Carrier Sense Multiple Access with Collision Detection (CSMA/CD) that enables transmitting nodes to detect a collision. After detecting a collision, a transmitting node waits a random delay time (Backoff algorithm) and then attempts to re-transmit the message. If the node detects a collision again, it waits for a period randomly chosen between 0 and twice as long time to try to re-transmit the message. Thus, we cannot predict how long the collision resolution will take. This randomness is the main limitation of using Ethernet in the Real-Time systems.

On the other side, one of the most important characteristics required by a real-time system is predictability. Predictability can partly be met by ensuring that all the timing constraints (deadlines) are met and all actions need to be time bounded. In the Backoff algorithm these constraints are not satisfied thus, predictability can not be assured. In order to assure predictability collisions have to be avoided and one way to achieve this is to make a real-time network protocol over Ethernet that provides a best-effort service only.

The report firstly presents the research laboratory and then it describes more precisely the context of this project leading to requirements. Afterwards, it shows the results of the analysis of our proposed protocol including its design. Finally, it describes implementation of the designed protocol, measurement results obtained and conclusions drawn at the end of this project.
2. CONTEXT

The SAN group is implementing the shared data space system **GSpace** and for its real-time extension, **RGSpace**, a real-time network protocol over Ethernet is needed.

We will first present GSpace and RGSpace and then we will describe the main features of real-time systems.

2.1. GSpace

GSpace is an implementation of a distributed shared data space system. The shared data space model was introduced in the coordination language Linda [3]. In Linda, processes communicate exchanging data through the shared data space using the simple yet powerful set of operations. The data elements that circulate in the shared data space are called *tuples*. A tuple is a sequence of typed fields with a specified value.

A simple way to implement a shared data space is that of employing a centralized architecture deployed in a single node. However, this design has two main drawbacks: firstly the single node represents a single point of failure and secondly the node might become the bottleneck of the entire system.

To circumvent these issues, GSpace was designed as a distributed system running across a set of nodes. Yet, its design is such that its distribution is completely transparent to the application processes. Figure 1 shows the deployment of several GSpace kernels across a battery of interconnected nodes. Each kernel provides the infrastructure for storing a *slice* of the data space. GSpace provides to application processes a global view of the data space although each process binds its local GSpace kernel.

![Figure 1: GSpace kernels deployment in a set of nodes](image-url)

A plethora of implementations for distributed shared data space exists in literature. A common pitfall of those systems is the use of a single static system-wide strategy for distributing tuples across the network of nodes. Often this distribution strategy is dictated by either the application domain or the underlying hardware. As a consequence, those systems are not flexible enough to be employed in environments for which were not designed. Moreover, it could be the case that within the same application the distribution needs may differ for different tuple types.

GSpace provides a suite of several distribution strategies (such as store on the local slice, replicate to all slices, caching tuples retrieved remotely) that can be extended with the introduction of new strategies. In GSpace tuples are typed (actually a tuple is implemented as a Java object) and each tuple type is distributed according to a
specific distribution strategy. In addition, GSpace monitors and evaluates the application usage pattern for a given tuple type during run-time. In doing so, GSpace detects when the actual distribution strategy for a tuple type is not efficient. Whenever this happens, GSpace identifies which is the best strategy and switches to it on the fly, without stopping the application execution.

A more detailed description of GSpace and its features can be found in [4].

2.2. RGSpace

While GSpace is well suited for regular systems, it lacks necessary properties for real-time systems. Handling requests is done in a best-effort fashion, giving no time guarantees. For hard real-time systems, it is essential for tasks to finish before the given deadline. GSpace's best-effort handling is clearly unacceptable. Therefore, this group worked on an extension to GSpace that does have timing properties. We call this extension RGSpace.

The next figure shows the separation of concerns in RGSpace.

![Figure 2: Separation of concerns in RGSpace](image)

Apart from the functional specifications and distribution requirements, RGSpace also takes into account timing constraints and timing error handling policies. The former declare periods of requests and deadlines whereas the latter specify what must happen if a deadline is missed or a task is unschedulable. Examples of timing error handling policies could be shut down the system, reject the task, or reschedule the task with a less strict deadline.

Similar to distribution policies, both timing constraints and timing error handling policies are converted into a timing constraints descriptor and an error handler descriptor respectively, which are downloaded into RGSpace kernels at booting time.

While GSpace has only one thread for handling requests in a best-effort manner, the timing constraints in RGSpace cannot permit such a strategy. Instead, RGSpace has several threads to handle one request at a time. The threads are controlled by a scheduler that preempts threads when necessary.
2.3. Real-Time systems
A real-time system is one in which it is possible to predict and control when computations take place. In real-time applications, the correctness of a computation depends not only upon its results, but also upon the time at which its outputs are generated. The measures of merit in a real-time system include:

**Predictably fast response:**
The system should respond quickly and predictably to urgent events.

**High degree of schedulability:**
The timing requirements of the system must be satisfied even at high degrees of resource usage.

**Stability under transient overload:**
When the system is overloaded by events, and it is impossible to meet all the deadlines, the deadlines of selected critical tasks must still be guaranteed.

In their pursuit of all of these objectives, real-time systems make use of a variety of scheduling algorithms. These methods of analysis allow engineers to assign priorities to different tasks, then spread the tasks out to ensure that the ones at the highest priority levels always meet their deadlines. [5]
3. REQUIREMENTS

Description
The goal of this project is to design and implement a real-time protocol over Ethernet that will be used by the real-time distributed shared data space RGSpace.

![System architecture](image)

**Figure 3: System architecture**

Preference will be given to designs that are situated on top of the network stack. In other words, those designs that do not require modifications of the standard network protocols. The reason is that such protocols are more independent from the operating systems.

There is just one application running in each node and applications always send messages periodically (an application can skip a period and send a message in the next one).

The protocol must be reliable and it must deliver any message within its deadline or be discarded otherwise; therefore the protocol must take latency and deadlines into account.

Messages are either sent from one node to another (point-to-point) or from one to many (broadcast).

The network is an isolated subnet (not connected to other networks) and it is exclusively reserved for the real-time traffic; it means that this is the only protocol running through the network.

The system is allowed to break and this protocol does not need to solve it.
**Constraints**

**Network speed:**
Either 10 Mbps or 100 Mbps

**Message deadline:**
10 ms shortest

**Message period for each node:**
100 ms shortest

---

**Example:**
Message deadline = 15ms  
Message period = 100ms

---

**Figure 4:** Deadline example

**Size of the message received from RGSpace and delivered to RGSpace:**
8 Kbits (1 KB) maximum, 960 bits (120 B) average and 8 bits (1 B) minimum of pure data (roughly)

**Programming language:**
C++ or Java

The number of nodes should be not more than 10.
**RGSpace interface**

RGSpace will call the following functions:

- **SendMessage** (dataLength: Integer, deadline: Integer, receivers: Integer, data: Char*): Integer
  - receivers:
    - Broadcast = 0
    - Unicast = Identification of the node
- **ReceiveMessage** (dataSize: Integer*, data: Char*): int

RGSpace will provide the following parameters in a configuration file:

- Required Shortest Deadline
- Required Shortest Period

Therefore, we have two deadlines: the shortest deadline that RGSpace can require and then a specific deadline for each message.

**Optional extensions**

The protocol should be extendable to deal with:

- Message priorities:
  - High priority (system administration)
  - Low priority (data distribution)
- Non-periodic messages.
- Crashes.
- A generic interface to switch among different standard networks, in this way the RGSpace will be transparent to the network.
- More than one application running in each node.
4. ANALYSIS

4.1. Requirements analysis
The shortest deadline is 10 ms, every node must be able to transmit messages at least every 10 ms.

The longest message transmission time is 0.8 ms and it happens under the worst conditions:
- lowest network speed: 10 Mbps
- biggest message size: 8 Kbits

The maximum message propagation delay for different network speed is:
- 10 Mbps: 51.2 μs
- 100 Mbps: 5.12 μs
  They are negligible.

We will first study the existent real-time protocols over Ethernet and we will compare them with our requirements in order to get some ideas for our design.
4.2. Summary of existing real-time protocols over Ethernet

We will explain several existing protocols and each protocol will be presented in the following way:

- Introduction
- Network Protocol
- Network Topology
- Communication protocol/architecture
- Details
- Evaluation (we will specially pay attention on protocols on top of the network stack)
  - Good sides
  - Bad sides

4.2.1. RTPS: Real-Time Publish-Subscribe

Introduction:
Real-Time Innovations, Inc. (RTI) is a Stanford University spin-off in real-time tools and distributed network architectures. They provide a real-time Publish-Subscribe networking middleware, called Network Middleware for Distributed Real-Time Applications (NDDS). [6]

NDDS is network middleware that simplifies the development of distributed real-time applications. It provides application developers with a publish-subscribe model and it uses the RTPS protocol.

Figure 5: NDDS middleware
**Network protocol:**
It is placed on top of the network stack
RTPS uses UDP/IP for all of its communications.

**Network technology:**
Ethernet

**Communication protocol/architecture:**
Publish-Subscribe

**Details:**

- **Delivery reliability / Memory usage / Determinism**
The RTPS protocol allows programmers to trade-off time determinism, delivery reliability, and memory usage. Publishers declare that they are capable of reliable transmission by specifying a maximum number of messages they are willing to store in a history cache for retries. Subscribers declare that they want reliable service and specify how many messages will be stored to assemble in-order packets. The reliable subscriber middleware acknowledges the receipt of each packet. The RTPS protocol retries dropped packets until the message is received, the allocated buffers are exhausted, or a subscription-specific deadline expires.
The buffer size on both ends determines the number of times the middleware retries for dropped packets. Thus, greater allocation of memory results in more reliable operation, up to a large number of retries if desired.

- **Latency Probability Formula**
CSMA/CD is clearly not deterministic because a back off algorithm is used; we cannot determine how long the collision resolution will take. Nonetheless, considerable experience with Ethernet shows that the unpredictable latency simply is not a factor in most real-time designs.

RTI provides a formula which shows that the CSMA/CD back off algorithm can perform so efficiently as to provide 99% probability that packets will be delivered.

With this formula, given an acceptable probability of error, we can find how long we can expect delay-free operation for a given packet size, transmission rate, and deadline. The results are encouraging. For instance, we can say that on a 100 Mbit Ethernet network, sending 1000 128-byte packets per second, you can be 99% sure that there will not be a delay of more than 1 millisecond caused by collisions in about 1140 years.
On the other side, if the deadline is 0.5 ms, the 99% assurance is reduced to 9 years. The performance degrades rapidly with loading and shorter delays. On a 10Mb net, we can only be 99% sure of no delay of 2ms with 1000 128-byte packets per second (about 1/10th the theoretical bandwidth) for 1.3 hours. This goes up to 293,000 years on a faster 100 Mb network.

<table>
<thead>
<tr>
<th>Bandwidth (Mbits/sec)</th>
<th>Packet Size (bytes)</th>
<th>Message Rate</th>
<th>Tmax (ms)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>128</td>
<td>1000</td>
<td>2</td>
<td>293K yrs</td>
</tr>
<tr>
<td>100</td>
<td>128</td>
<td>1000</td>
<td>1</td>
<td>1140 yrs</td>
</tr>
<tr>
<td>100</td>
<td>128</td>
<td>1000</td>
<td>0.5</td>
<td>9 yrs</td>
</tr>
<tr>
<td>100</td>
<td>1024</td>
<td>1000</td>
<td>2</td>
<td>604 yrs</td>
</tr>
<tr>
<td>100</td>
<td>1024</td>
<td>1000</td>
<td>1</td>
<td>2 yrs</td>
</tr>
<tr>
<td>100</td>
<td>128</td>
<td>5000</td>
<td>1.5</td>
<td>483 yrs</td>
</tr>
<tr>
<td>10</td>
<td>64</td>
<td>1000</td>
<td>7</td>
<td>9 yrs</td>
</tr>
<tr>
<td>10</td>
<td>64</td>
<td>1000</td>
<td>2</td>
<td>10 hrs</td>
</tr>
<tr>
<td>10</td>
<td>128</td>
<td>250</td>
<td>4</td>
<td>2 yrs</td>
</tr>
<tr>
<td>10</td>
<td>128</td>
<td>1000</td>
<td>7</td>
<td>1 yr</td>
</tr>
<tr>
<td>10</td>
<td>128</td>
<td>1000</td>
<td>2</td>
<td>1 hr</td>
</tr>
<tr>
<td>10</td>
<td>128</td>
<td>500</td>
<td>3.5</td>
<td>53 days</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>1000</td>
<td>8</td>
<td>23 yrs</td>
</tr>
</tbody>
</table>

**Latency Assurance:** This table contain examples from the formula presented showing how long you can be 99% sure that the network will run without incurring the specified delay \(T_{max}\) under different loading conditions. Most applications can run comfortably on a fast 100Mbit net. Many will work on a 10Mbit net if a delay of 2ms or so is acceptable.

**Figure 7:** RTI formula’s results

**Evaluation:**
RTPS has an unusual way to handle real-time characteristics, every node sends messages when it is needed and we hope that there are no collisions.

Anyhow, if we compare these results with our requirements, we can see in the precedent table that for 10 Mbps, 1024 bytes packet size, rate of 100 messages and 8 ms deadline, there will not be any collisions in 23 years. Therefore, these results satisfy our requirements, even in the worse case (in our requirements the deadline is 10 ms). Nevertheless, in these results it is not specified the amount of nodes that send packets. It is a significant parameter, because if we increase the amount of nodes the protocol has to deal with more packets.

- **Good sides:**
  - The protocol is placed on top of the network stack.
  - The formula allows us to determine in advance the probability of success full transmission taking into account our requirements (network speed, message size, message rate and deadline) and the results are very promising.

- **Bad sides:**
  - RTI provides literature about the NDDS middleware from a marketing point of view. This literature mixes concepts and does not give clear description about how the middleware works.
4.2.2. RTCAST: Lightweight Multicast for Real-Time Process Groups

Introduction:
In this protocol, a distributed system is structured as a group of cooperating processes using two key primitives: group membership and fault-tolerant multicast communication. [7]

RTCAST is a real-time group communication protocol that provides multicast and membership service for real-time process groups which may exchange periodic and aperiodic messages.

The service supports message transport within a deadline, atomicity (message is delivered to either all processes or to none at all), and order for multicasts within a group of communicating processes in the presence of processor crashes and communication failures.

Network protocol
It is implemented on top of the IP protocol, and sends messages using broadcast or IP multicast if available. If neither of these services is available, RTCAST can resort to multiple point-to-point transmissions to send each message to the group.

The following figure shows different modules, we will describe each module next.

![Diagram of RTCAST architecture](image)

**Figure 8**: RTCAST architecture
Network technology:
Ethernet

Communication protocol/architecture
Peer-to-peer, broadcast or multicast

Details

➢ Platforms
It is designed to run on out-of-the-shelf, non-real-time operating systems and hardware, and has been implemented for Solaris, Linux, Windows NT, and OpenGroup MK.

➢ RTCAST
RTCAST is implemented as a set of event handlers which respond to different events. These events include message reception and timeouts (for example the token timeout). Each event handler is implemented as a separate thread which services events of a particular type.

There are two primary event handling threads in RTCAST which have real-time requirements. The first is the thread responsible for receiving messages from the network. This receive thread is responsible for copying incoming messages into a protocol buffer, and performing the message ordering and failure detection checks. This thread is also responsible for sending queued messages during the token possession, and sending the heartbeat message to pass the token to the next process.

The second thread is the thread which handles timeouts, such as the token timeout which is used to detect failures in the token ring protocol. RTCAST implements a timeout service where registered timeouts are placed in a queue. A timeout thread is responsible for periodically checking the queue and executing the event handler for any timeout which has expired. RTCAST is structured as a library which is linked to the application. This means that each process in an RTCAST group runs its own instance of the protocol stack and operates independently of other RTCAST processes, even ones residing on the same physical host.
Logical Token Ring Protocol

RTCAST utilizes a logical token ring algorithm to regulate access to the network.

![Logical Token Ring Protocol Diagram]

- Shows the order of senders on the logical ring.
- −→ Traces the flow of some multicast message m1.

**Figure 9:** RTCAST logical token ring protocol

Processes are ordered by the protocol such that each group member has a well-known predecessor and successor on the ring. Message sequence numbers are used to ensure a total delivery order and detect communication failures (if a process was expecting \( s \) next, and it receives \( s+1 \) instead, it knows it has missed one). To ensure atomicity, if a process does not receive a message and the message cannot be retransmitted before its deadline, that process is removed from the group. To ensure timeliness, each process \( Pi \) has a maximum token hold time \( Ti \). A token holder releases the token when it has sent all of its messages, or when \( Ti \) has expired, whichever comes first. This guarantees a bounded token rotation time, for a group of size \( n \) with a total message propagation delay \( \Delta \).

\[
T_{token} = \sum_{i=1}^{n} (T_i + \Delta)
\]

The TRT is important for message admission control and schedulability analysis because it determines how often a process will be allowed to send messages periodically.

It also determines the timeout used to detect token loss and process failures. If the token is lost or a process fails, its successor will time out waiting for the token. The successor will then update the group membership and generate a new token.

Admission Control and Schedulability Analysis

This module implements a protocol layer above the RTCAST layer, which performs schedulability analysis to regulate traffic flow.

RTCAST is unaware of the type of admission control policy used, therefore it allows use of a number of admission control policies to ensure timeless while leaving consistency to the multicast algorithm.
Clock Synchronization
The clock synchronization is needed to implement message order, because in arbitrary networks messages may be received out of order by the machine.

This module provides a synchronization service using a probabilistic algorithm.

**Experiment 1: Base Token Rotation Time (BaseTRT)**
The next figure shows the BaseTRT for RTCAST group sizes of one node to four nodes. This is the time to rotate the token without sending any data messages, providing an indication of the overhead of token handler processing and transmitting the token to next node. The data shows that each node introduces approximately one half millisecond for token processing. The data was collected by averaging over 1000 token rotations.

<table>
<thead>
<tr>
<th>Group size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseTRT</td>
<td>0.43</td>
<td>1.16</td>
<td>1.72</td>
<td>2.30</td>
<td>2.84</td>
<td>3.39</td>
</tr>
</tbody>
</table>

*Figure 10: RTCAST’s BaseTRT results (ms)*

**Experiment 2: Throughput**
Throughput measurements were done varying the number of 1-byte data messages sent by each node on each token possession and measured the aggregate number of messages sent. Small messages were used so that the aggregate measurement represents the maximum message transmission rate achievable. The data for several group sizes are shown in following figure. Again each data point represents an average over 1000 token rotations.

*Figure 11: RTCAST’s maximum message transmission rate using 1-byte messages*

The aggregate throughput is not dependent on group size, indicating that the protocol does not introduce additional overhead in the data path when more members are added.
Experiment 3: Message latency

To determine RTCAST protocol latency, varying the number of (1-byte) messages sent by each node per token visit and measure the elapsed time between subsequent releases of the token. This gives an indication of the overhead incurred by data messages waiting for the token to arrive while other group members transmit and receive messages in atomic, total order. Latency measurements are presented for several group sizes in the next figure. Note that the latency increases as more messages are sent per token round due to the increase in token rotation time when each member holds the token for a longer time. Similarly, the latency grows with the number of group members since adding members also increases the token rotation time.

![Figure 12: RTCAST’s Data message latency with 1-byte messages](image)

Evaluation:
Three experiments are presented in the RTCAST documentation and we will compare those experiments with the worst case of our requirements:

- Network speed: 10 Mbps
- Messages deadline: 10 ms
- Message size: 8 Kbits (1 KB)
- Total message period for each node: 100 ms

We can calculate that the message transmission time is 0.8 ms (8 Kbits / 10 Mbps) and that the message period for each node is 10 messages per second.

Evaluating BaseTRT:
If we take a group of 6 nodes, we see that the BaseTRT is 3.39 ms. To satisfy the deadline (10 ms), we can think that the token visits each node at least every 10 ms, then the TRT is 10 ms. It means that we have 6.61 ms (10 – 3.39) to send messages in every token rotation.

Let’s think that every node has a message to send every 100 ms, but all nodes have decided to send a message at the same time (worst case). It means that in one token rotation all messages must be sent (in the next 90 ms will not be any messages). We know that a message is transmitted in 0.8 ms and we have 6.61 ms to transmit 6 messages. We work out that it is feasible \((6.61 - 6 \times 0.8)\), even we have 1.8 ms left. Hence, these results satisfy our requirements.
Evaluating Throughput:
Our message transmission period is 100 messages per second and per node, therefore it is easily satisfied.

Evaluating message latency:
We know that the token must visit each node at least every 10 ms to satisfy the deadline and that the message period is 100 ms per node. Thus, every time that the token visits one node there will not be more than one message to be sent.

The last figure shows that for 1 message per round and for a group of 4 nodes, the latency is approximately 3.5 ms, it is much smaller than our deadline. Thus, it satisfies our requirements.

Admission control and schedulability analysis module:
In our case, RGSpace implements the admission control and schedulability analysis, thus the same idea is implemented above our network protocol.

Good sides:
- The protocol is placed on top of the network stack.
- Results are very promising.
- The logical token passing protocol gives design flexibility; depending on the requirements we can design a mechanism or another one, whichever works better.

Bad sides:
- The TRT increases when the number of nodes increases and consequently we get worse deadlines.
4.2.3. Real-Time Ethernet Network at Home

Introduction:
It is a research about a home network which provides both real-time and non-real-time capabilities for one coherent, distributed architecture. It is based on a new type of real-time token protocol that uses scheduling to achieve optimal token-routing in the network, visiting only those nodes needing attention. The first objective of this network is to be a network for entertainment, control and information, as for streaming media. [8]

Network protocol:
The token protocol is situated between the network and link layer of the OSI model. Therefore, it requires modifications of the standard network protocols

![Architecture Diagram](image.png)

Figure 13: Architecture

Unlike other protocols, the token does not follow a fixed path, visiting every node during each rotation. Instead, the token is scheduled and follows a dynamic route, visiting only those nodes needing attention. When needed, a node may reschedule the token and give it a new route through the network. This happens when new connections are added, or existing connections are changed in the network. Before applying the new schedule, the node will check if this schedule is feasible. Token management needs special attention to make this type of network robust. Special measures have to be taken if the token is lost (e.g. the device that holds the token is removed), or if multiple tokens exist.

Network technology:
IEEE 1394 (Firewire), Bluetooth or Ethernet

Communication protocol/architecture:
Peer-to-peer and broadcast

Details:
- **Token**
To make sure only one stream is being sent at the time, a token is used. This token contains all information needed to make scheduling decisions. When a token arrives at a node, the node looks in the token to determine which stream is up for transmitting its data (one node can have different streams). It also determines the duration of time the stream may transmit; this is called the token holding time. When this time has elapsed, the scheduler
looks in the token to determine which node holds the next stream that is to be scheduled, and then sends the token there.

- **The scheduler**
  The scheduler is decentralized and resides in every node.
  In order to schedule real-time traffic on a network, scheduling algorithms are used, which are originally meant for scheduling tasks on a processor.

- **Add new streams**
  Adding new streams to the network can only be done by the node holding the token, so every node should get the token regularly. Tests proof that is better to define a real-time stream for adding new streams, so it is guaranteed that every node gets an equal share of the time.

The validity of the bandwidth reservation of existing streams is guaranteed by letting nodes, that wish to add a stream, perform a feasibility analysis first. This feasibility analysis determines whether the network will not be overloaded, so bandwidth guarantees can be maintained.

- **Add new nodes**
  Adding new nodes to the network is done by regularly broadcasting messages to the network, indicating nodes wishing to join should reply to this broadcast. When they do, they will be added to the list of nodes in the token, making sure they will get the token after some time, so they too can add real-time streams.

- **Real-Time / Non-Real-Time**
  In order to make sure a non-real-time traffic gets a piece of the total network bandwidth available, only a portion of the total bandwidth is allocated as usable for real-time traffic.
  When non-real-time traffic is being transmitted, the token uses a simple round-robin strategy to travel across the network, visiting every node.

- **Token loss**
  To prevent token loss, a monitoring node is introduced: the node that held the token last. When a node sends the token to the next node, it becomes the monitor. The previous monitor stops monitoring. When the node holding the token dies, the monitor will notice and send the token to the node the deceased node should have sent the token to.

- **Time synchronization protocols**
  Different computers have a different sense of time (all clocks are not equal) and time synchronization protocols are needed to solve the timing problems between nodes.

**Evaluation:**

- **Bad sides:**
  o It requires modifications of the standard network protocol.
  o It takes into account non-real-time traffic and it is not our requirement.
  o It is more appropriated for media streams.
4.2.4. RT-EP: Real-Time Ethernet Protocol

Introduction:
RT-EP is basically a token-passing protocol in a bus and it has been designed to avoid collisions in Ethernet media. [9]

Network protocol:
The protocol has been implemented in GNU/Linux, directly over the data link layer. Therefore it requires modifications of the standard network protocols.

Each station has a transmission queue, which is a priority queue where all the packets to be transmitted are stored in priority order. Each station also has a set of reception queues that are also priority queues. Packets with the same priority are stored in FIFO order. The number of reception queues can be configured depending on the number of application threads (or tasks) running in the system and requiring reception of messages. Each application thread should have its own reception queue attached. The application has to assign a number, the channel ID, to each application thread that requires communication through the protocol. This channel ID is used for the purpose of identifying communication endpoints in a given station.

The network is logically organized as a ring. Each station knows which other station is its predecessor and its successor, so the logical ring can be built. The protocol works by rotating a token in this logical ring. The token holds information about the station having the highest priority packet to be transmitted and its priority value. The network operates in two phases:

- **Phase 1: the priority arbitration**
The token travels through the whole ring, visiting all the nodes. Each station checks the information in the token to determine if one of its own packets has a priority higher than the priority carried by the token. In that case, it changes the highest priority station and associated priority in the token information; otherwise the token is left unchanged. Then, the token is sent to the successor station. This process is followed until the token arrives at the *token_master* station, finishing the arbitration phase.

- **Phase 2: the transmission of an application message**
The *token_master* station sends a message to the station with the highest priority message, which then sends the message. The receiving station becomes the new *token_master* station.

Network technology:
Ethernet

Communication protocol/architecture:
Peer-to-peer and broadcast
Details:

- **Platforms**
  Implemented in MaRTE OS, but also suitable for other Real-Time OS.

- **Implementation and modeling**
  This protocol offers three functions to any application using the network: *send_info* (to send a message), *recv_info* (to receive a message), and *init_comm* (to initialize the network). The application threads encapsulate the information in a message type, which is used both for transmission and reception. This message type contains the destination station address, the destination channel ID, and the priority of the message. When a message is sent, it is stored in the priority queue on the transmitting station. There is only one thread, the *Main Communication Thread*, that is responsible of reading the packets from the transmission queue and of writing the received packets into the reception queues. The highest priority packet to be transmitted determines the priority used for the priority arbitration token.

![](image)

**Figure 14:** Functionality and details of RT-EP topics

In order for this protocol to work, the maximum number of communicating threads running in the system must be known at configuration time. This is the usual case in this kind of real-time system.
State machine

RT-EP can be described as a state machine for each station in order to understand its functionality and obtain the relevant parameters for the different operations involved in the timing model. The following picture shows the states and the transitions between them, which are shortly described next:

- **Offline.**
  It is the starting state reached during configuration time. Each station reads a configuration file describing the token ring and gets configured as one of its stations. The station configured as the initial token_master is set to the Send_Initital_Token state and the others are set to the Idle state.

- **Idle.**
  The station listens for the arrival of any packet.
  When a packet is received:
  - if it is an Info Packet the station switches to the Recv_Info state
  - if it is a Token Packet, two different states can be reached:
    - Send_Info (if a Transmit Token is received)
    - Check_Token (when a regular Token is received).

- **Send_Initial_Token.**
  The station reaching this state becomes the token_master. A token is sent to the successor station, and the current station switches to the Idle state.

- **Check_Token.**
  If the station isn't the token_master the Send_Token state is reached; if it is, it switches to the Send_Info state if the Station Address is the current station, or to the Send_Permission state when not.

- **Send_Token.**
  The station compares the priority of the token with the highest priority element on its transmission queue, updates the token if its own priority is higher, and sends the token to next station. Then it switches to the Idle state.
- **Send_Permission.**

  The token_master role is lost and the Transmit Token is built and sent to the highest priority station.

- **Send_Info.**

  This is the state in which a station has the highest priority packet on the ring and it is allowed to transmit it.

- **Recv_Info.**

  The information is written into the appropriate reception queue and the station switches to the Send Initial Token state, becoming the token_master.

**Evaluation**

- **Good sides:**
  - Message priority is an optional extension of the requirements and this protocol takes it into account.

- **Bad sides:**
  - It requires modifications of the standard network protocol.
4.2.5. RETHER

Introduction:
RETHER features a hybrid mode of operation that automatically switches a local area network between the token passing RETHER mode for real-time and the CSMA/CD mode for non-real-time at the same time. [10]

Network protocol:
RETHER is placed over the data link layer; therefore it requires modifications of the standard network protocols

It is completely transparent to existing network protocols such as TCP/UDP and IP. Consequently, all existing network applications can continue to run on a RETHER network without any changes. New real-time applications have to be specifically written against the API provided by RETHER, which is based on the standard socket interface.

RETHER provides a set of procedural interfaces for applications to reserve network bandwidth, and once they are admitted, it guarantees the reservations throughout the lifetime of the applications. Under RETHER, individual applications can send a certain amount of data \( D \) in each periodic cycle \( T \). The amount of data is specified in the reservations and thus varies from application to application. The length of the cycle is fixed but can be changed at the system initialization time. To start a RETHER connection, which is *uni-directional*, the sender application makes a library call, `reservation()`, which, upon successfully creating a RETHER connection, returns a socket descriptor. From that time on, the sender application can `send()` the reserved amount of data through the returned socket descriptor, and RETHER ensures that the data be delivered at the rate \( D / T \). The receiver application simply receives the sent data using normal `receive()` calls without making any special arrangements.

Network technology:
Ethernet

Communication protocol/architecture:
Peer-to-peer

Details:

- **Real-Time / Non-Real-Time:**
  In the absence of real-time connections, the RETHER network operates under the original CSMA/CD protocol. When the first real-time bandwidth reservation request arrives, the RETHER network switches to a *token-passing* mode. In this mode, channel access for *all* traffic, real-time and non-real-time, is regulated by a token. Intuitively time is divided into cycles. In each cycle, the token first services all the nodes that have real-time data to send (called *real-time (RT)* nodes), and then it attempts to service *non-real-time (NRT)* nodes in a round-robin fashion. Essentially NRT nodes share the bandwidth remaining in each cycle after all RT nodes have been serviced. Note that *all* network nodes, including those making bandwidth reservations, are NRT nodes. The network stays in the token-passing mode until the last real-time session terminates. At this time, the network switches back to the CSMA/CD protocol. This hybrid scheme reduces the performance impact due to token passing on non-real-time traffic.
Token Passing

The token circulates in cycles so that real-time connections can access the network periodically. Because RETHER does not assume a globally synchronized clock, the token cycle time is maintained as a counter called the residual cycle time in the token itself. At the beginning of each cycle, the residual cycle time is set equal to a full token cycle time. When the token visits a node, the node subtracts its token-holding time from this counter. Once the residual cycle time reaches zero, the token is passed back to the first real-time node and a new cycle begins. The following figure shows an example token visit schedule within a token cycle.

![Token Passing Figure](image)

Figure 16: Token passing example

Node 1, 3, and 6 are real-time nodes, each of which is assumed to have a token holding time of 6 ms every time the token visits them in the real-time mode. As every network node has non-real-time data to send, ALL network nodes are non-real-time nodes. In this example, the token visits real-time nodes in the first 18 ms of the 33 ms token cycle, and then it visits non-real-time nodes in the rest of the cycle in a round robin fashion. Note that the token continues the visiting schedule for non-real-time nodes in the next cycle (Node 6) from where it left off in the previous cycle (Node 5).

The token-holding time at each node is based on the amount of data the node is allowed to send. For real-time nodes, the bandwidth reservation determines the amount of data that needs to be sent out during every cycle. For non-real-time nodes, the amount of data that can be sent out is limited by the total unreserved bandwidth and the size of messages in their output queues. RETHER has incorporated a mechanism to ensure that all the nodes use the unreserved bandwidth fairly. When the token visits a node, if it does not have data to send, it merely hands the token to its successor after subtracting the time to process the token.

Fault-Tolerance

REThER incorporates a built-in fault tolerance mechanism to ensure continued network operation despite token loss due to machine failures, or token corruption due to random bit errors. Each RETHER node is required to monitor the health of its successor in the token passing schedule. When a node N sends the token to its successor S, it starts an acknowledgment timer waiting to hear from S. If S is alive, it sends an acknowledgment back to N when it sends the token forward to its own successor. If the successor node is dead for some reasons, the timer at the monitoring node times out and it pings the successor to ensure that the successor indeed dies. This extra ping is necessary to check if the successor is still alive but actually drops the token due to reasons like bit errors. On detecting a failure, the monitoring node broadcasts a message announcing the failure and regenerates a new token.
Single/Multi segment

RETHER has been extended to operate across multi-segment switched Ethernet environment.

Evaluation

Bad sides:

- It requires modifications of the standard network protocol.
- It is based on real-time and non-real-time traffic bandwidth reservation and non-real-time traffic is not our requirement.
4.2.6. Adaptive Traffic Soothing

Introduction:
The traffic smoother first gives real-time packets priority over non-real-time ones in order to eliminate contention within each local node. Second, it smooths a non-real-time stream to reduce collision with real-time packets. [11]

This traffic smoothing can decrease the packet-collision ratio on the network. The traffic smoother, installed at each node, regulates the node’s outgoing non-real-time stream to maintain a certain traffic-generation rate. In order to provide a reasonable non-real-time throughput, the traffic-generation rate is allowed to adapt itself to the underlying network load condition.

Network protocol:
This traffic smoother requires a minimal change in the OS kernel without any modification to the current standard of Ethernet MAC protocol or the TCP or UDP/IP stack.
It is installed between the UDP or TCP/IP layer and the Ethernet MAC layer, and works as an interface between them.

Network technology:
Ethernet

Communication protocol/architecture:
Peer-to-peer, broadcast and multicast

Details:

Platforms
Linux and Windows NT platforms

Evaluation:

Bad sides:
- It requires modifications of the standard network protocol.
- It is based on smoothing non-real-time stream and non-real-time is not our requirement.
### 4.3. Comparison of existing protocols and choice

We will first summarize the performance characteristics in the next table and then we will explain our choice.

<table>
<thead>
<tr>
<th>Network Protocol</th>
<th>Network Technology</th>
<th>Communication Protocol / Architecture</th>
<th>Good sides</th>
<th>Bad sides</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTPS</td>
<td>-Over transport layer</td>
<td>-Ethernet</td>
<td>-Publish-Subscribe</td>
<td>-On top of the network stack -Results are promising</td>
</tr>
<tr>
<td>RTCAST</td>
<td>-Over transport layer</td>
<td>-Ethernet</td>
<td>-Peer-to-peer -Broadcast -Multicast.</td>
<td>-On top of the network stack -Results are promising -Design flexibility</td>
</tr>
<tr>
<td>Real-Time Ethernet Network at Home</td>
<td>-Over data link layer</td>
<td>-Ethernet -Firewire -Bluetooth</td>
<td>-Peer-to-peer -Broadcast</td>
<td>- Better efficiency</td>
</tr>
<tr>
<td>RT-EP</td>
<td>-Over data link layer</td>
<td>-Ethernet</td>
<td>-Peer-to-peer -Broadcast</td>
<td>-Message priorities are supported</td>
</tr>
<tr>
<td>RETHER</td>
<td>-Over data link layer</td>
<td>-Ethernet</td>
<td>-Peer-to-peer</td>
<td>-It requires modifications of the standard network protocol -It takes into account non-real-time traffic and it is not our requirement</td>
</tr>
<tr>
<td>Adaptive Traffic Soothing</td>
<td>-Over data link layer</td>
<td>-Ethernet</td>
<td>-Peer-to-peer -Broadcast -Multicast.</td>
<td>-It requires modifications of the standard network protocol -It takes into account non-real-time traffic and it is not our requirement</td>
</tr>
</tbody>
</table>

Figure 17: Performance characteristics
In the requirements, preference is given to designs that are situated on top of the network stack. In other words, those designs that do not require modifications of the standard network protocols. Thus, we will first concentrate in the first two protocols, RTPS and RTCAST.

Although results are very promising with RTPS, it has an unusual way to handle real-time characteristics and the literature mixes concepts from a marketing point of view. In addition, RTPS does not take into account message priorities for instance and this design is not flexible to deal with this issue since every node sends messages when it is needed.

RTCAST provides also good results and the logical token passing protocol gives design flexibility. Hence, we can design our token passing protocol depending on our requirements. In this way, the new design has flexibility to be extended.

Therefore, we will take the RTCAST protocol as a base for our protocol design. However, RTCAST provides services that we do not need, e.g. membership service. For this reason, we will customize RTCAST to satisfy our requirements.
5. DESIGNING TRIP

In designing our protocol we aimed to use of-the-shelf hardware and software. The reasons for choosing Ethernet are already mentioned at the beginning of this document. Following the same criterion, we designed a protocol on top of the network stack avoiding modifications of the standard network protocols.

However, Ethernet cannot provide any time guarantees about message delivery because we cannot predict how long the collision resolution will take place. For this reason, in our protocol the access to a shared medium is controlled using a logical token. The logical token turns around the logical ring according to a logical token passing algorithm. This algorithm gives flexibility to the design facilitating modifications or extensions.

In this chapter we describe the design of our Token based Real-time Light Protocol over Ethernet (Trip).
5.1. Architecture

Trip uses broadcast communication and it is situated on top of the network stack; therefore modifications of the standard network protocols are not required.

![Context architecture](image18.png)

**Figure 18**: Context architecture

In order to control the access to the medium, we use a logical token on top of the network stack. The logical token turns around the logical ring in a round robin fashion and just the node that holds the token can broadcast messages.

The following picture shows Trip’s architecture.

![Trip’s architecture](image19.png)

**Figure 19**: Trip’s architecture
In the next figure we have a local network composed of 4 nodes. Every node is aware of the network architecture and they know which their predecessor and successor nodes are. This introduces limitations that nodes cannot just come and go, the system has to restart. When the token is broadcasted, every receiver node verifies whether the receiver is itself or not.

In our example, Node 1 broadcasts the token and Node 2 holds it (phase 1). Afterwards Node 2 broadcasts two data messages (phase 2). Finally, Node 2 broadcast the token and Node 3 holds it (phase 3).

**Figure 20:** A broadcasts the token message and B holds it
Figure 20 shows the internal modules of the Trip software architecture. RGSpace interacts with Trip through the \textit{Api}. The Api provides methods for sending and receiving messages. RGSpace sends and receives messages through Trip asynchronously. For this reason, Trip is provided with two message buffers: the \textit{Sending Buffer}, for storing messages to be sent, and the \textit{Receiving Buffer}, for storing received messages.

The \textit{Network Manager} is the active module of Trip. The Network Manager is associated with a thread that continuously probes the \textit{Receiver} for incoming messages. The Receiver listens to the network stack for incoming messages. Upon receiving a message, the Receiver de-marshals the message and passes it to the Network Manager. Messages can be of two types: token and data message.

When the broadcasted token is received by all the nodes in the network, the Network Manager in each node checks if it is its turn to send data messages. If not, the token is ignored. Otherwise, the Network Manager will send all pending messages in the Sending Buffer through the \textit{Broadcaster}. For each sending message, the Broadcaster marshals the message and broadcasts it. After all pending messages are sent, the Network Manager sends the token to the next node in a logical ring.

If the received message is a data message then the Network Manager stores the message in the Receiving Buffer.
**Message ordering**
Since there is just one node sending messages at any time, every node receives messages in the same order. Every message contains a Sequence Number and it is a global identification number.

### 5.2. Message types
In this section we describe the message types used in Trip.

![Different message types](image)

**Figure 22:** Different message types

**Token:**
The logical token turns around the network in a round robin fashion and just the node that holds the token can broadcast data messages. It is just a normal message with a specific header.

- **Message type:**
  It specifies the message type, Token or Data. In this case is Token.

- **Sequence number:**
  It is a global identification number for each message that gives information about message ordering.

- **Sender:**
  It is the identification of the sender node.
- **Receivers:**
  Even all messages are broadcasted; this parameter is used for message processing. It specifies the receivers and it can represent logical broadcast or logical unicast communication:
  - **Broadcast = 0**
  - **Unicast = Identification of the node**
  The Token is sent to the next token holder, thus, it is always logical unicast communication.

- **Data length:**
  The token has no data, then, it is 0.

**Data:**
Data messages are broadcasted to transmit data among the nodes.

- **Message type:**
  It specifies the message type, Token or Data. In this case is Data.

- **Sequence number:**
  It is the global identification number for each message.

- **Sender:**
  It is the identification of the sender node.

- **Receivers:**
  Even all messages are broadcasted; this parameter is used for message processing. It specifies the receivers and it can represent logical broadcast or logical unicast communication:
  - **Broadcast = 0**
  - **Unicast = Identification of the node**

- **Data length:**
  It is length of the data. It is used for message processing.

- **Data:**
  It is the data that must be transmitted among the nodes.
5.3. **Logical token ring algorithm**

In this section the algorithm for the logical token ring is presented.

⇒ If my node is the node 1
   ♦ Broadcast token (Token holder is node 2)

⇒ Repeat for ever
   ♦ Receive message (blocking)
   ♦ Expected Sequence Number ++
   ♦ If message is Token
     • If Receiver is my node
       ▪ While there is at least one Data message in the Sending Buffer
         ○ Broadcast Data message
         ○ Receive (blocking)
         ○ Expected Sequence Number ++
         ▪ Broadcast Token (Token holder is next node)
   ♦ If message is Data
     • If Receiver is me
       ▪ Store it in Receiving Buffer
5.4. Class diagram
This section provides a view of UML Class Diagram of Trip.

The arrow means instantiates

Figure 23: Class diagram
5.5. Analytical parameter representation
In this section we introduce the basic mathematic concepts that are used for calculating the values of the parameters to guarantee a correct real-time behavior of Trip.

5.5.1. Definitions

**THT:**
*Token Holding Time* delimits how long each node holds the token.

**TRT:**
*Token Rotating Time* delimits how long the token needs to rotate.

**BaseTRT:**
*Base Token Rotating Time* is the time that the token needs to rotate without sending any data messages (receive, process and broadcast the token).

**HighestLoadTRT:**
*Highest Load Token Rotating Time* is the time that the token needs to rotate while every node sends the biggest data message (1024 B). We can say that it is the longest turn that can happen. In other words, it is the worst case; every node wants to send the biggest message at the same time.

**N:**
It is the number of nodes.

**BiggestLatency:**
*Biggest Message Latency* is the time it takes for the biggest message (1024 B) to cross over the network. One Ethernet packet can contain 1500 B of data, thus the biggest message can be sent in one Ethernet packet.

\[
\text{Overhead} + 1024 < 1500
\]

**MessageDeadline:**
*Message Deadline* is the deadline for one specific message. Every message has its own deadline specified by RGSpace.

**SupportedShortestDeadline:**
*Supported Shortest Deadline* is the shortest deadline that Trip can provide to RGSpace. It depends on the network and nodes performances (network speed, hub/switch speed, processors, operating system...)

**RequiredShortestDeadline:**
*Required Shortest Deadline* is the shortest deadline that RGSpace will require for a message (specified in a configuration file)

**SupportedShortestPeriod:**
*Supported Shortest Period* is the shortest message period for one node that Trip can provide to RGSpace. It is the shortest interval between two messages.

**RequiredShortestPeriod:**
*Required Shortest Period* is the shortest message period for one node that RGSpace requires in the configuration file.
**RequiredMaxNumMsgPerTurnPerNode:**
RGSpace specifies the RequiredShortestDeadline and the RequiredShortestPeriod. From these two values we can calculate the Required Maximum Number of Messages Per Turn and Per Node:

\[
\text{RequiredMaxNumMsgPerTurnPerNode} = \frac{\text{RequiredShortestDeadline}}{\text{RequiredShortestPeriod}} \text{ (up rounded)}
\]

**SupportedMaxNumMsgPerTurnPerNode:**
Supported Maximum Number of Messages Per Turn and Per Node is the maximum number of messages that can be sent per turn, per node supported by Trip.
5.5.2. Real-Time guarantees

Since this protocol is used by real-time systems, it has to provide real-time guarantees to meet message deadlines. At the same time, we try to optimize resource usage.

In order to be as accurate as possible and take into account message processing time, we conducted some tests and measured the values that we are interested in. This phase is what we call “protocol profiling”. The profiling phase is done every time that Trip is booted in a different context (different number of nodes, different hub/switch, different network characteristics...). We are interested in measuring the following parameters:

- BaseTRT
- HighestLoadTRT

RGSpace specifies in a configuration file RequiredShortestDeadline and RequiredShortestPeriod. Before booting Trip, these values must be checked to make sure that Trip can handle them. The constraints are:

- RequiredShortestDeadline \( \geq \) SupportedShortestDeadline
- RequiredShortestPeriod \( \geq \) SupportedShortestPeriod

Where SupportedShortestDeadline and SupportedShortestPeriod are the shortest deadline and the shortest period that Trip can support, respectively.

Now, we will explain each constraint based on BaseTRT and HighestLoadTRT (measured values).

**RequiredShortestDeadline \( \geq \) SupportedShortestDeadline**

RGSpace determines a MessageDeadline for each message and each message must be delivered within its MessageDeadline. On the other side, RequiredShortestDeadline is the shortest MessageDeadline that RGSpace may require.

The logical ring architecture establishes a relation between deadlines and TRT. The logical token must visit each node at least every RequiredShortestDeadline. Hence, the TRT is bounded to the RequiredShortestDeadline. During the profiling phase, we measure the TRT with maximum load messages. During the token rotation, each node broadcasts the biggest message. We consider then the worst case scenario, meaning that we assume as SupportedShortestDeadline the HighestLoadTRT, which is the longest turn that the protocol experienced during the profiling phase:

\[
\text{SupportedShortestDeadline} = \text{HighestLoadTRT}
\]

Thus, we can say that:

\[
\text{RequiredShortestDeadline} \geq \text{HighestLoadTRT}
\]
**RequiredShortestPeriod >= SupportedShortestPeriod**

We want to express this constraint in terms of *RequiredShortestDeadline*, *BaseTRT* and *HighestLoadTRT*.

The *HighestLoadTRT* assumes that just one single data message is sent each time the node holds the token. However, when the system is deployed the THT could be long enough for sending more than one data message. The extra amount of time depends on the difference between *RequiredShortestDeadline* and *HighestLoadTRT*. The bigger the difference, the longer is the extra time for sending more data messages.

For optimize resource usage, it is desirable that each node broadcasts more than one data message during each THT. Actually, we wanted to send more messages with 1 token, thus avoiding sending extra messages (tokens) and wasting bandwidth. This can save the burden of broadcasting the token each time a data message has to be transmitted.

From the previous constraint, we have that *RequiredShortestDeadline* must always be larger than *HighestLoadTRT*. We could fix the TRT to be equal to *HighestLoadTRT* and be sure that all real-time constraints are met. However, if we want to optimize, we have to *slow down* the protocol for gaining larger THT and being able to send more than one data messages. For this reason, we fix the TRT equals to *RequiredShortestDeadline*:

\[
(1) \quad \text{TRT} = \text{RequiredShortestDeadline}
\]

We calculate TRT in the following way:

\[
(2) \quad \text{TRT} = \text{BaseTRT} + (\text{THT} \times N)
\]

where THT is used to broadcast only data messages.

The latency for broadcasting biggest message is *BiggestLatency*, thus THT is equal to the following:

\[
\text{THT} = \text{BiggestLatency} \times \text{SupportedMaxNumMsgPerTurnPerNode}
\]

If we replace THT in (2):

\[
(3) \quad \text{TRT} = \text{BaseTRT} + (\text{BiggestLatency} \times \text{SupportedMaxNumMsgPerTurnPerNode} \times N)
\]

According to (1), TRT is equal to the *RequiredShortestDeadline*. Then, if we replace TRT in (3), we will get the following expression:

\[
(4) \quad \text{RequiredShortestDeadline} = \text{BaseTRT} + (\text{BiggestLatency} \times \text{SupportedMaxNumMsgPerTurnPerNode} \times N)
\]

The latency for sending the biggest data message can be calculated according to the following:

\[
\frac{\text{HighestLoadTRT} - \text{BaseTRT}}{N} = \text{BiggestLatency}
\]

If we replace *BiggestLatency* in (4) and extract *SupportedMaxNumMsgPerTurnPerNode*:

\[
(5) \quad \text{SupportedMaxNumMsgPerTurnPerNode} = \frac{\text{RequiredShortestDeadline} - \text{BaseTRT}}{\text{HighestLoadTRT} - \text{BaseTRT}}
\]
HighestLoadTRT – BaseTRT
In general, we can calculate the period for sending messages (per node) dividing the time needed for a token turn by the number of messages that each node can send during the token holding time. Thus we can have the following formula:

$$\frac{TRT}{SupportedMaxNumMsgPerTurnPerNode}$$

If we substitute (1) in (6) then we have the following:

$$\frac{RequiredShortestDeadline}{SupportedMaxNumMsgPerTurnPerNode}$$

Now, let’s replace (5) in (7) and we obtain that:

$$\frac{RequiredShortestDeadline}{RequiredShortestDeadline – BaseTRT}$$

We transform into:

$$\frac{RequiredShortestDeadline (HighestLoadTRT– BaseTRT)}{RequiredShortestDeadline – BaseTRT}$$

Finally we obtain:

$$\frac{RequiredShortestDeadline (HighestLoadTRT– BaseTRT)}{RequiredShortestDeadline – BaseTRT}$$

All the variables on the right-hand side of the last equation are specified in the profile file. Based on this equation Trip should give a GSpace an answer if their message with a specific deadline can be transmitted or not.
6. IMPLEMENTATION AND RESULTS

6.1. Operating System
The operating system used by RGSpace is Timesys Linux/Real-Time [5].

The standard Linux is limited as an operating system for real-time systems:
- Limited number of fixed priority levels
- No support for priority inheritance
- Limited QoS (Quality of Service) support
- Lack of support for high-resolution timers
- No support for periodic tasks
- Potentially non-preemptible kernel with possibly long system calls
- Limited support for non-desktop systems

TimeSys Linux users benefits from such features as:

- **Real-time Linux applications:**
  Any Linux process can now become a real-time process. You are no longer constrained to choose between a real-time OS and Linux. You do not have to embed a thin real-time OS layer below the Linux kernel.

- **POSIX (Portable Operating System Interface) support for your real-time needs:**
  TimeSys Linux provides complete support for the traditional real-time systems paradigm of using a fixed-priority preemptive scheduling policy. In fact, it supports 2048 priority levels. It also supports priority inheritance on mutexes to avoid the unbounded priority inversion problem.

- **QoS (Quality of Service) delivery:**
  TimeSys Linux provides direct and explicit support for QoS delivery to your real-time applications using the notation of CPU and network reservations (also called reserves).

- **Real-time support for legacy applications:**
  A convenient feature is that you can take existing legacy applications running on Linux and endow them with QoS guarantees.
6.2. Compiler
As programming language we used C++ and GNU G++ as compiler. Although RGSpace is written in Java and compiled in native code using jRate (Java Real-Time Extension [12]), incompatibilities between jRate and network libraries prevented us to use Java for Trip.

It means that Java and C++ codes must interoperate. It can be done using Compiled Native Interface (CNI). The Java code is compiled to native code by jRate and then linked to the Trip C++ object by CNI.

![Compilation Diagram]

Figure 24: Compilation

6.3. Implementation
During the implementation of Trip, the biggest problem was that C++ Standard Library does not handle sockets, timers, and threads. There are other class libraries available, but using other libraries we lose control over the code and it can bring expensive costs to real-time requirements, processing times for instance. Thus, we decided to use C Standard Library and transform the required functions into C++ code.

6.4. Results
Several tests were done in order to measure Trip’s performance. We will first present the testing context and the results of each test. We will finally present the integration of Trip into RGSpace.
6.4.1. Testing context

The tests were done under the following features:
- Switch: 100MB
- Operating System: Timesys Linux
  - Red Had Linux 7.3
  - Kernel 2.4.7 – Timesys 3.1.214
- Nodes:
  - Pentium III
  - 256 MB RAM
  - 535 MHz

6.4.2. Tests

Trip has a profiling phase and it must be executed every time that Trip will run in a different context (different number of nodes, different hub/switch…). The profiling provides the shortest deadline and the shortest message period that Trip can support.

In the configuration file, the user can specify the number of turns for the profiling. For example, we tested with 1 million turns. Therefore, the token will turn 1 million times and each turn is measured. The longest turn, the shortest turn and the average are registered.

The results show that the shortest turn and the average are approximately same, but the longest turn is usually much longer than the average. It means that there are a few turns (up to 1 million turns) that are much longer than the others. We call them “peaks”.

**Why do the peaks happen?**

The profiling phase takes around 14 minutes with 1 million turns. During this time, Trip is using all CPU resources, but the operating system must also take care of other system tasks. Then, Trip has to wait for some time and peaks happen.

We will show the tests that we carried out with different number of nodes. We will see that in every test there are two or three peaks up to 1 million turns and the longest peak among all tests takes 8.184 ms.

For each test we will present the following files:

- **Configuration file**
  The user must provide this file before running Trip. It contains the settings that will be used by Trip.

- **Profiling file**
  Trip creates this file at the end of the profiling phase. It contains the values that Trip can support (deadline, period)

- **BaseTRT detail file**
  Trip creates this file during the profiling phase and it contains detailed information about BaseTRT (shortest turn, longest turn, average, peaks…)

- **HighestLoadTRT detail file**
  Trip creates this file during the profiling phase and it contains detailed information about HighestLoadTRT (shortest turn, longest turn, average, peaks…)
Two nodes

- **Configuration file**

TRIP Config File

Node:
1

Number of nodes:
2

Port:
2222

Required Shortest Deadline (in milliseconds):
10

Required Shortest Period (in milliseconds):
20

Number of turns for profiling:
1000000

TRIP mode (run=0, test=1, debug=2):
0

---

- **Profiling file**

TRIP Profiling File

SupportedShortestDeadline (ms):
0.824254

SupportedShortestPeriod (It depends on RequiredShortestDeadline) (ms):
0.661029

INFORMATION:
RequiredShortestDeadline: 10ms
RequiredShortestPeriod: 20ms
BaseTRT: 0.174778ms
HighestLoadTRT: 0.824254ms
SupportedShortestDeadline is calculated by:

- RequiredShortestDeadline (HighestLoadTRT - BaseTRT)
- RequiredShortestDeadline - BaseTRT

The highest peak on HighestLoadTRT was 5.02ms
Higher deadlines than this value will have more probability to be met.
There were 2 peaks longer than 2ms, up to 1000000 turns.
**BaseTRT detail file**

DETAILED RESULTS BaseTRT
Number of profiling turns: 1000000
Expected sequence number: 2000001

Global results:
Start time: 1094115681::162.38
End time: 1094115860::574.263
Elapsed time: 179412ms

Partial results:
Longest turn: 8.184 ms
Shortest turn: 0.168 ms
Average: 0.174778 ms

PEAKS
Peak 1:
  Turn duration: 6.583ms
  Global time (since beginning): 10447.1ms

Peak 2:
  Turn duration: 8.184ms
  Global time (since beginning): 34625.2ms

**HighestLoadTRT detail file**

DETAILED RESULTS HighestLoadTRT
Number of profiling turns: 1000000
Expected sequence number: 4000001

Global results:
Start time: 1094115860::574.784
End time: 1094116689::574.784
Elapsed time: 829074ms

Partial results:
Longest turn: 5.02 ms
Shortest turn: 0.683 ms
Average: 0.824254 ms

PEAKS
Peak 1:
  Turn duration: 5.02ms
  Global time (since beginning): 30209.5ms

Peak 2:
  Turn duration: 4.35ms
  Global time (since beginning): 60208.6ms
Three nodes

Configuration file

TRIP Config File

Node:
1

Number of nodes:
3

Port:
2222

Required Shortest Deadline (in milliseconds):
10

Required Shortest Period (in milliseconds):
20

Number of turns for profiling:
1000000

TRIP mode (run=0, test=1, debug=2):
0

Profiling file

TRIP Profiling File

SupportedShortestDeadline (ms):
1.23558

SupportedShortestPeriod (It depends on RequiredShortestDeadline) (ms):
1.00215

INFORMATION:
RequiredShortestDeadline: 10ms
RequiredShortestPeriod: 20ms
BaseTRT: 0.259427ms
HighestLoadTRT: 1.23558ms

SupportedShortestDeadline is calculated by:

- \( \text{RequiredShortestDeadline} (\text{HighestLoadTRT} - \text{BaseTRT}) \)
- \( \text{RequiredShortestDeadline} - \text{BaseTRT} \)

The highest peak on HighestLoadTRT was 6.013ms
Higher deadlines than this value will have more probability to be met.
There were 2 peaks longer than 3ms, up to 1000000 turns.
BaseTRT detail file
DETAILED RESULTS BaseTRT
Number of profiling turns: 1000000
Expected sequence number: 3000001

Global results:
Start time: 1094119547::271.998
End time: 1094119811::359.722
Elapsed time: 264088ms

Partial results:
Longest turn: 3.787 ms
Shortest turn: 0.252 ms
Average: 0.259427 ms

PEAKS
Peak 1:
   Turn duration: 3.787ms
   Global time (since beginning): 33531.2 ms

HighestLoadTRT detail file
DETAILED RESULTS HighestLoadTRT
Number of profiling turns: 1000000
Expected sequence number: 6000001

Global results:
Start time: 1094119811::360.255
End time: 1094121051::360.255
Elapsed time: 1.24039e+06ms

Partial results:
Longest turn: 6.013 ms
Shortest turn: 1.094 ms
Average: 1.23558 ms

PEAKS
Peak 1:
   Turn duration: 6.013ms
   Global time (since beginning): 33956ms

Peak 2:
   Turn duration: 5.532ms
   Global time (since beginning): 34444.1 ms
### Four nodes

- **Configuration file**

TRIP Config File

Node:

1

Number of nodes:

4

Port:

2222

Required Shortest Deadline (in milliseconds):

10

Required Shortest Period (in milliseconds):

20

Number of turns for profiling:

1000000

TRIP mode (run=0, test=1, debug=2):

0

---

- **Profiling file**

TRIP Profiling File

SupportedShortestDeadline (ms):

1.64864

SupportedShortestPeriod (It depends on RequiredShortestDeadline) (ms):

1.34991

INFORMATION:

RequiredShortestDeadline: 10ms

RequiredShortestPeriod: 20ms

BaseTRT: 0.34535ms

HighestLoadTRT: 1.64864ms

SupportedShortestDeadline is calculated by:

\[
\text{SupportedShortestDeadline} = (\text{HighestLoadTRT} - \text{BaseTRT}) / (\text{RequiredShortestDeadline} - \text{BaseTRT})
\]

The highest peak on HighestLoadTRT was 6.872ms

Higher deadlines than this value will have more probability to be met.

There were 3 peaks longer than 4ms, up to 1000000 turns.

---

- **BaseTRT detail file**
DETAILED RESULTS BaseTRT
Number of profiling turns: 1000000
Expected sequence number: 4000001

Global results:
Start time: 1094122735::337.326
End time: 1094123085::353.886
Elapsed time: 350017ms

Partial results:
Longest turn: 5.648 ms
Shortest turn: 0.337 ms
Average: 0.34535 ms

PEAKS
Peak 1:
   Turn duration: 5.648ms
   Global time (since beginning): 5467.63ms


highestLoadTRT detail file
DETAILED RESULTS HighestLoadTRT
Number of profiling turns: 1000000
Expected sequence number: 8000001

Global results:
Start time: 1094123085::354.43
End time: 1094124738::354.43
Elapsed time: 1.65347e+06ms

Partial results:
Longest turn: 6.872ms
Shortest turn: 1.522ms
Average: 1.64864ms

PEAKS
Peak 1:
   Turn duration: 6.215ms
   Global time (since beginning): 30449.8ms

Peak 2:
   Turn duration: 6.872ms
   Global time (since beginning): 32973.9ms

Peak 3:
   Turn duration: 4.182ms
   Global time (since beginning): 60450.2ms
### 6.4.3. Trip integration into RGSpace

RGSpace and Trip projects have been carried out in parallel. Intensive testing has been conducted for both applications in stand-alone mode. As a next step, we had foreseen the integration of the two applications and testing. Unfortunately, we could start the integration phase only few days before this report was completed. Therefore, in these pages we can only describe some premature results due to the integration part.

During the integration, we noticed that the operating system scheduler was not scheduling RGSpace’s and Trip’s threads in the same way. RGSpace’s threads were set with a priority, period, cost and deadline. Trip’s thread (Network Manager) was set just with a high priority. We suppose that once Trip’s thread was created and lost its processing time (e.g. when waiting for a message from the network), the operating system did not reschedule it any more.

We did not have time to make the required changes in our codes, but we thought about a possible adaptation and we will explain it.

We can create a thread in RGSpace and then this thread can call Trip’s Network Manager. In this way, RGSpace has the whole control over all threads, even over Trip’s thread, and it can specify priorities, periods, costs and deadlines.

RGSpace tasks and Trip are scheduled periodically as the following figure shows:

![Figure 25: Periodical execution of Trip and RGSpace](image)

When the operating system executes Trip, Trip rotates the token just once per period and in this rotation every node broadcasts all data messages stored in its sending buffer.

Moreover, all nodes must be well synchronized. Thus, a synchronization mechanism is required.
7. CONCLUSIONS

In this report we presented the design decisions that led to the implementation of a real time protocol over the Ethernet. We named our protocol Trip. We tested Trip as stand-alone application, and the results were promising. In a network composed by two nodes, the shortest deadline that Trip provides for a message is 0.8 milliseconds and it increases 0.4 milliseconds per added node. Unfortunately, due to the lack of time, we are not able to present the results of tests where Trip is integrated into RGSpace.

Designing and implementing real-time applications require refining our notion of time, getting used to work with small time units, such as milliseconds, microseconds or even nanoseconds.

A suggestion for future work is that proposed protocol can be improved by taking utilization and efficiency of the system into account, simplifying or avoiding synchronization, facilitating scalability and making possibilities for transient node behavior.
8. REFERENCES

[1] Eindhoven University of Technology
http://w3.tue.nl/nl/

http://www.win.tue.nl/san/

Generative Communication in Linda
1985
http://portal.acm.org/citation.cfm?id=2433

Dynamic Adaptation of Data Distribution Policies in a Shared Data Space System
Proc. Int’l Symp. On Distributed Objects and Applications (DOA), Larnaca, Cyprus
October 2004
http://www.win.tue.nl/~grussell/

http://www.timesys.com/

[6] RTI Real-Time innovations
- Can Ethernet be Real-Time?
  - NDDS DataSheet
  - NDDS Product Brief
  2004

RTCAST: Lightweight Multicast for Real-Time Process Groups
1997

Scott Johnson, Farnam Jahanian, Akihiko Miyoshi, Dionisio de Niz, and Ragunathan Rajkumar
Constructing Real-time Group Communication Middleware Using the Resource Kernel
1999 or later
www.eecs.umich.edu/~scottdj/papers/rtss00.ps

A Real-Time Ethernet Network at Home
Distributed and Embedded Systems group Faculty of Computer Science - University of Twente
2002 or later
http://www.ub.utwente.nl/webdocs/ctit/1/0000009c.pdf
[9] José María Martínez, Michael González Harbour and J. Javier Gutiérrez
- A Multipoint Communication Protocol based on Ethernet for Analyzable Distributed Real-Time Applications
Departamento de Electrónica y Computadores - Universidad de Cantabria
2002 or later

[10] Tzi-cker Chiueh
RETHER: A Software-Only Real-Time Ethernet for PLC Networks
1999

Achieving Real-Time Communication over Ethernet with Adaptive Traffic Smoothing
2000
http://csdl.computer.org/comp/proceedings/rtas/2000/0713/00/07130090abs.htm

[12] jRate (Java Real-Time Extension)
http://www.cs.wustl.edu/~corsaro/jRate/index.html