QuAL: Quality Assurance Language

(Thesis Proposal)

Patrícia Gomes Soares Florissi

Advisor: Prof. Yechiam Yemini

Technical Report CUCS-007-94

Abstract

Distributed multimedia applications are sensitive to the Quality of Services (QoS) provided by their computing and communication environment. For example, scheduling of processing activities or network queueing delays may cause excessive jitter in a speech stream, rendering it difficult to understand. It is thus important to establish effective technologies to ensure delivery of QoS required by distributed multimedia applications.

This proposal presents a new language for the development of distributed multimedia applications: Quality Assurance Language (QuAL). QuAL abstractions allow the specification of QoS constraints expected from the underlying computing and communication environment. QuAL specifications are compiled into run time components that monitor the actual QoS delivered. Upon QoS violations, application provided exception handlers are signaled to act upon the faulty events. Language level abstractions of QoS shelter programs from the heterogeneity of underlying infrastructures. This simplifies the development and maintenance of multimedia applications and promotes their portability and reuse. QuAL generates Management Information Bases (MIBs) that contain QoS statistics per application. Such MIBs may be used to integrate application level QoS management into standard network management frameworks.

Summary

In these pages, we highlight the main contributions of the work described in the Thesis Proposal. The current version of the proposal is a long document because it includes a comprehensive set of examples written in the language we are defining (QuAL) and an appendix section that makes the document self-contained. The appendix contains a summary of the language QuAL extends (Concert/C), the complete syntax of QuAL, and a description of the runtime system of QuAL.

1 The Problem

- In this work we address the problem of developing technologies to monitor and control Quality of Services (QoS) for distributed multimedia computing and communication applications.
- It is of practical significance for such applications to be able to define (e.g., end-end delay of 100ms for a voice transmission), monitor (e.g., check the actual end-end delay), and control (e.g., drop the service because delay is too high) QoS-related events occurring in a distributed system.
- In order to be able to express and monitor certain QoS measures, real-time language constructs must be provided.
- Automation of QoS management information collection eases QoS monitoring and control.
- System management entities can greatly improve system performance by being able to access data gathered from application-level QoS monitoring and by managing the system in an application-customized manner.
- We address the problem of handling QoS-related events by defining Quality Assurance Language (QuAL), a new programming language that explicitly incorporates constructs and data types to abstract such events occurring in a distributed system.
- Abstractions at the language-level provide a common platform to shelter heterogeneity at lower levels (e.g., OS and transport-layer), especially with regard to QoS provision facilities, and enable a higher level of functionality (e.g., compile-time and run-time type checking of QoS demands of the communicating applications).
- To minimize the inertia overhead of using a new language and of porting existing applications, QuAL consists of a concise set of language constructs that extend an existing process-oriented language (Concert/C).

2 QuAL: Quality Assurance Language

- QuAL abstracts a distributed system as a set of processes that exchange information using communication streams *with associated QoS demands* (e.g., tolerable loss rate, bandwidth, etc.).
- QuAL's abstractions can be incorporated as an extension to any process-oriented language, Concert/C being our choice for the first QuAL prototype.
- In order to abstract QoS on communication streams, QuAL provides means to:
    1 Specify QoS measures and processing constraints of message streams.
    2 Access temporal properties (sending, arrival, and processing times) of messages and prioritize message processing accordingly.
    3 Specify control of QoS and processing of message streams via attached QoS and processing exception handler constructs.
    4 Automatically generate Management Information Bases (MIBs) that contain traces gathered from QoS and processing monitoring.
- By providing communication QoS at the language level, QuAL *bridges heterogeneity at lower communication-service providers* (e.g., transport layer, OS).
- By enabling the generation of MIBs, QuAL *promotes management that focus on application properties* rather than on infrastructure.

2.1 Specification of QoS Measures and Processing Constraints

- QuAL provides constructs for the specification of QoS measures that define network-level (e.g., propagation delay) and application-level (e.g., delivery rate) properties that streams being communicated should satisfy.
- QuAL is unique in providing a strongly-typed language-level abstraction for QoS measure specification that shelters heterogeneity at lower layers (e.g., transport-layer and OS).
- Both sender and receiver processes participate in the QoS attribute negotiation, as opposed to existing mechanisms where either the sender or the receiver chooses the QoS of the communication while the other needs to comply with the choice (e.g., MCAM).
- QoS measures are used by the runtime to reserve communication and processing resources and to detect periods in which the required quality is violated.
- Real-time language constructs are added to provide means to express execution timing constraints (such as processing start time and deadline, hard or soft deadlines, etc.) necessary to express many QoS measures.

2.2 Real-time Ports

- QuAL defines real-time ports, that is, ports that allow the access to temporal properties (sending, arrival, and processing times) of messages crossing them.
- All manipulation of temporal properties is transparent to the user and lives at a lower layer (QuAL's runtime), similarly to the way that network-layer header processing is transparent to the transport layer.
- Message processing may be prioritized based on: (1) order of arrival (as in Concert/C) or (2) sending and arrival times.
- Existing language-level communication abstractions do not capture communication temporal properties.
- The communication abstraction in QuAL is completely backwards compatible with the communication abstraction in Concert/C.

2.3 Specification of QoS and Processing Control

- When the requested QoS is violated, QuAL's exception handling mechanism is responsible for calling upon proper corrective actions.
- If either the application or the underlying system violates the QoS measures of a stream, an exception message is sent to an user-defined exception handler port associated with the connection.
- Exception handler processes are bound to the exception handler port and take the proper user-defined corrective measure whenever messages arrive at the port.
- Exception handler functions can be defined for soft timing constraint violations.
- To deal with periods in which the required QoS is being violated, QoS parameter re-negotiation is possible during execution time.
- Exception handling mechanisms provide means to implement graceful recovery and degradation in a way best suited to the specific demands of the application.

2.4 Application Management Automation

- The data on the communication QoS provision used to monitor user-defined measures is also used by QuAL's runtime to generate application-customized Management Information Bases (MIBs) that contain traces gathered from QoS and processing monitoring.
- MIBs are customized in that they reflect peculiarities of the applications and allocated resources.
- QuAL's runtime follows the *Simple Network Management Protocol* (SNMP) standards for the generation of MIBs.

- Based on the MIBs, external management entities can control the quality of services provided by the underlying system (e.g., re-configuring the network to reduce the propagation delay).
- Applications, on the other hand, use the MIBs to control its own performance in relation to services being provided (e.g., as tuning the message processing rate based on the number of buffered messages).
- QuAL supports the specification of *traps*, a mechanism to asynchronously notify exceptional conditions recorded in the MIBs without polling them.
- Management is for the first time an integral part of the application (as opposed to being an integral part of the underlying system) because MIBs are structured and allocated as part of the development process of the system.

1 Introduction

In this work we address the problem of developing technologies to monitor and control Quality of Services (QoS) for distributed multimedia computing and communication applications. These emerging applications require distinct QoS in order to operate effectively. Applications such as radiology image processing for laser injections and monitoring of fluid dynamics for petroleum extraction require timely transfers of massive amounts of data for remote processing. These transfers can only be achieved by allocating corresponding *guaranteed amounts* of communication bandwidth and processing resources. Multimedia conferencing and remote shopping require interactions among participants that demand *strict end-end delays* (e.g., not higher than 100ms for audio transmissions) and *bounded jitter* or else interactions become unnatural. Virtual reality systems impose *application-dependent constraints* on the communication to bridge heterogeneity at the processing and communication levels. For example, a virtual world service provider may interact with a lower throughput terminal by restricting itself to send only every other video image. This reduction on the rate of data being transmitted enables a lower quality interaction.

It is thus of practical significance for such applications to be able to define (e.g., end-end delay of 100ms for a voice transmission), monitor (e.g., check the actual end-end delay), and control (e.g., drop the service because delay is too high) QoS-related events occurring in a distributed system. We use the multimedia conference example in Figure 1.1 to elaborate on the technical dimensions of the problem we are studying. In the figure, the components of the application are depicted in their respective layers which are separated using thick horizontal lines. The arrows depict information flow to and from the many peripheral devices displayed above the application layer. At the application-layer, participant A's voice and image are recorded in real-time by process S_A, generating two data streams. These streams and their interdependencies (synchronization) are stored in a local file system as well as played back (allowing a participant to see its own image). The streams are then broadcast over the network to the other participants using the underlying transport-layer services. At another site, process S_B records and broadcasts participant B's image and voice along with the English translation of his speech in the form of text annotations. Inter-dependencies among streams must be recovered when the information is received from the network at each side (necessary for lip-synchronization). Processes D_A and D_B are responsible for receiving and displaying the data streams from B and A, respectively.

QoS measures depend on the application being designed and must be defined accordingly. In the example illustrated in Figure 1.1, Sa and Db must firstly agree on the rate and the quality of the data exchanged (e.g., Sa cannot send 2 Mbps of HiFi Audio if Db is only capable of processing 64 Kbps of standard audio). In this case, application-level rate is the QoS measure. Sa and Db must also allocate processing resources to meet strict deadlines in sampling and reproducing the audio and video streams at the rates agreed (e.g., 8KHz and 30 video frames/sec, respectively, for standard sampling [campbell93]). The processes must also allocate connections to transmit the data at an appropriate rate and within certain delay bounds (e.g., a standard audio transmission requires a bandwidth of 64Kbps whereas a video transmission requires 25Mbps [campbell93]; the maximum end-to-end delay and inter-message delays tolerated for both connections in a real-time transmission can be of 250ms and 10ms, respectively). In this case, bandwidth, end-end delay, and inter-message delay are QoS measures. To provide lip-synchronization, the generation, transmission, and display of video messages must be interleaved with the ones of audio messages. Furthermore, the inter-dependencies between the messages must be captured and preserved. The QoS measure in this case is synchronization.

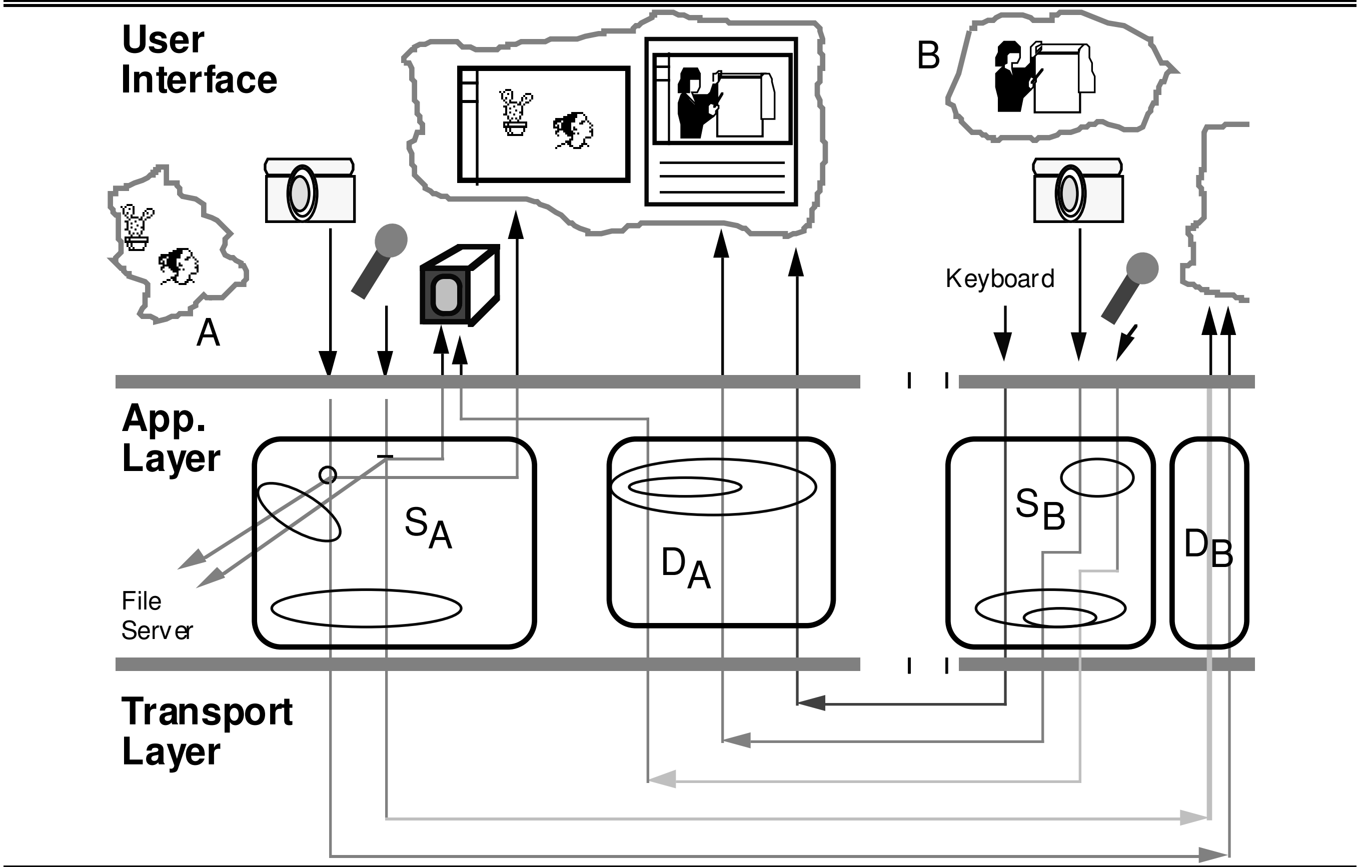

Figure 1.1: Multimedia Conference

Although QoS are negotiated, they must be tightly managed in an application-dependent manner to enable fast and specialized recovery and graceful degradation from QoS violations. There are QoS measures at the network level (e.g., delay), at the OS level (e.g., processing scheduling), and at the application level (e.g., message generation rate and synchronization) that need to be negotiated and managed between the network-service provider, the underlying OS and the application processes. Consider the arrival and display process of video messages described in Figure 1.2, where time is measured in ms since the start of the application. The inter-message display period (playout jitter) of a video stream is the main display quality factor and must be controlled to prevent a slow motion effect (e.g., playout jitters bigger than 67ms [campbell93]) or a fast-forward effect (e.g., playout jitters smaller than 33ms [campbell93]) on the display. Playout jitter must be controlled based on the message arrival process, to smooth any delay that may occur on the display caused by late arrivals. This control is illustrated in Figure 1.2 as follows. The display only starts after four messages arrive. The second message is displayed 33ms after the first one is displayed. However, since the fifth message has not arrived by the time the third should be displayed (1640ms+33ms), the display of the third message is delayed incurring a playout jitter of 46ms (to absorb 13ms of the probable delay on the display of the 5th message). Similarly, the display of the fourth is delayed and incurs a playout jitter of 67ms (absorbing 34ms). To preserve the notion of motion, playout jitter cannot exceed 67ms. Since the fifth messages is not received, the fourth is replayed to replace it. If the fifth message arrives late, it must be discarded. The arrival time monitoring and playout jitter control illustrated can be performed by looking only at the temporal properties of the data, such as sending, arrival and processing times.

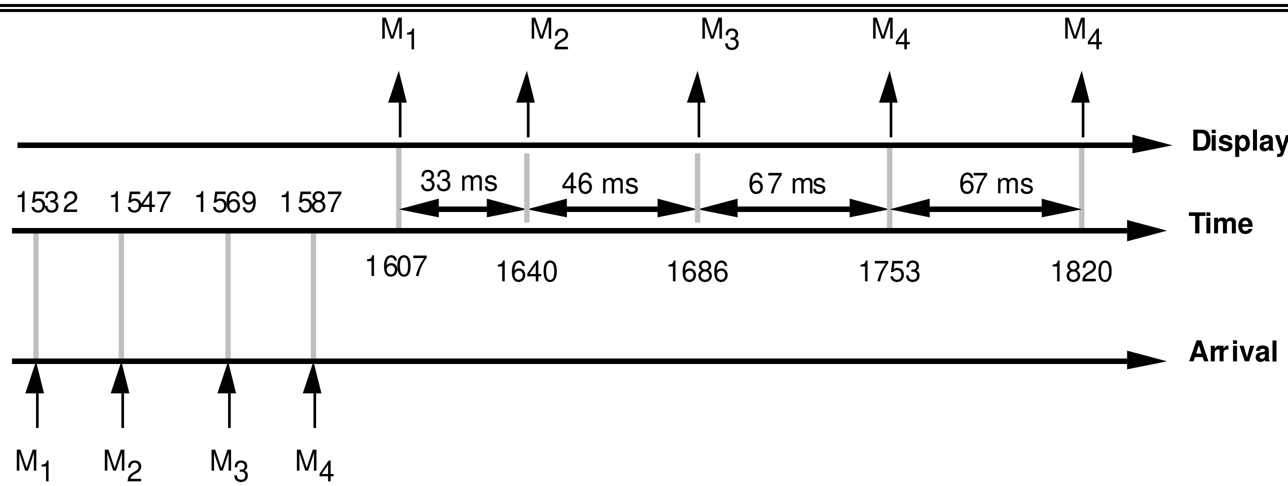

Figure 1.2: Arrival and Display Time of a Stream of Messages

A general framework to represent the negotiation and monitoring of QoS based only on temporal properties (sending, arrival and processing times) of communicated messages can be described as follows:

- let S represent the domain of streams where each stream consists of a set of messages and each message is represented by an index (that identifies the message) and its temporal properties;
- let qos: $S \rightarrow R$ be the function that given a stream returns the stream's value of the QoS being measured by qos (e.g., the average inter-arrival time of the messages in the stream);

- let qos^*: $S \rightarrow R$ be the function that given a stream returns the stream's *optimal* or *desired* value of the QoS being measured by qos (e.g., the average inter-arrival time that the messages in the stream should have had);
- let Δqos: $S \rightarrow R$ be the function that given a stream returns the QoS deviation of the stream (that is, given a stream s_i, $\Delta qos(s_i) = qos(s_i) - qos^*(s_i)$), and
- let β: $S \rightarrow R$ be a deviation bound function that given a stream returns an acceptable bound for its QoS deviation.

QoS negotiation consists of allocation of resources by all involved entities such that $\Delta qos(s_i) \leq \beta(s_i)$ for each communicated stream s_i and for each Δqos defined for the communication. QoS monitoring consists of verifying the compliance of these properties, causing exceptions to be raised whenever they are violated.

The quality of message arrival times in Figure 1.2 can be mapped into this QoS framework as follows. Let s_i represent the stream of messages since the transmission started until the transmission of the i-th message and $T_A(s_i)$ represent the arrival time of the i-th message ($T_A(s_i) = \infty$ if the i-th message is lost). In Figure 1.2, $T_A(s_1) = 1532$, $T_A(s_2) = 1547$, and so on. Furthermore, let $T_A^*(s_i)$ represent the *optimal* arrival time, that is, the time the i-th message should have arrived. For the video stream represented in the Figure 1.2, $T_A^*(s_i)$ can be expressed as a function of the display start time (1607ms) and the optimal playout jitter for display (33ms): $T_A^*(s_1) \leq 1607$, $T_A^*(s_2) \leq 1607 + 33$, $T_A^*(s_2) \leq 1607 + (2 * 33)$, etc. $\Delta T_A(s_i)$ represents the deviation between the actual and the optimal arrival times and must be bounded based on the buffer size and the actual display rate (to avoid buffer overflow and consequent loss due to too many message arrivals before their respective display time) . Let $B(s_i)$ represent the period of time the i-th message can be stored in the buffer which is computed based on the rate at which messages are arriving and being displayed ($B(s_i) = 0$ if the message arrives late or is lost). Relating to the QoS framework, qos is T_A, qos^* is T_A^*, B is β, and monitoring is employed to detect situations in which $\Delta T_A(s_i) > B(s_i)$.

Real-time language features and automation of management information collection are also needed to support QoS negotiation, monitoring and control. In order to be able to express and monitor certain QoS measures, real-time language constructs must be provided. For example, the synchronization between the audio and video streams can be expressed by specifying that a video message must be generated within 1/10sec of an audio message generation. Furthermore, if such deadlines are violated control must take place.

Applications management automation eases QoS monitoring and control. QoS statistics (e.g., average message processing rate, average message transmission delay, etc.) reflect the performance of the applications involved and of the underlying system services they use. QoS statistics can be automatically collected by automating the QoS monitoring process. This data can be used by the applications themselves to control their QoS. Furthermore, external resources management entities can access the QoS statistics to manage and improve the QoS provided by the underlying system. The information can be used as a metric for performing a more accurate distributed system resources management in a per-application basis.

In summary, the problem at hand demands technologies for the expression of QoS measures, monitoring, and control. To be complete, these technologies must be able to express real-time features. Finally, management entities can greatly improve system performance by being able to access data gathered from QoS monitoring.

Several approaches have been taken to handle these issues. New switching techniques [yemini93], transport-layer protocols [RFC1190], and session-layer [anderson90] algorithms have been proposed to deliver the QoS demanded by applications. New OS [govindam91] have been designed to properly allocate processing resources and schedule processes according to their timing constraints. Extensions to the OSI [osi] architecture [keller93] have been suggested to integrated the services provided at all layers to comply with the QoS demands. Ad-hoc approaches have been used in the development of earlier applications [fish89].

We address the problem of handling QoS-related events by defining the new programming language Quality Assurance Language (QuAL) that explicitly incorporates constructs and data types to abstract such events occurring in a distributed system. Abstractions at the language-level provide a common platform to shelter heterogeneity at lower service-provider layers (e.g., transport layer, OS layer). Emerging applications are inherently distributed and distributed systems tend to be heterogeneous, especially with regard to QoS provision facilities. For example, the ST-II [RFC1190] transport protocol has QoS parameters that differ from the ones available in XTP/PE [chesson88]. The design of QuAL and its runtime system aims at sheltering heterogeneity at the OS and transport layers. Furthermore, it allows the interoperability of an application written in QuAL with applications that use communication services offered at the representation-layer and application-layer. QuAL also includes real-time language features.

Another goal in the design of QuAL is the minimization of the inertia overhead of using a new language and of porting existing applications. Towards that goal, QuAL consists of a concise set of language constructs that extend an existing process-oriented language: Concert/C [auerbach92]. Concert/C, in turn, extends the C language with a small set of types and operators to integrate distributed processing abstractions.

QuAL provides means to automatically generate Management Information Bases (MIBs) that contain traces gathered from QoS and processing monitoring. Management is for the first time an integral part of the application development process as MIBs are structured and allocated as a result of the compilation process. The MIBs are defined according to the SMI [stallings93] standard and can be accessed by any management entity that follows the SNMP [stallings93] standard. Thus, the MIBs expose to the underlying management system the configuration of the applications currently running, integrating application monitoring into general network management.

This work is structured as follows. In Section 2, we present the language constructs introduced by QuAL. In Section 3, we discuss technologies needed to support QoS assurance and how they affect the implementation of QuAL. In Section 4, we show some examples written in QuAL. In Section 5, we discuss related work and place QuAL in this context. In Section 6, we conclude and present the thesis work schedule.

2 QuAL: Quality Assurance Language

QuAL abstracts a distributed system as a set of processes that exchange information using communication streams *with associated QoS demands* (e.g., tolerable loss rate, bandwidth, etc.). In this section, we describe the abstractions introduced by QuAL and give examples of how to use them.

QuAL's abstractions can be incorporated as an extension to any process-oriented language, Concert/C [auerbach92] being our choice for the first QuAL prototype. Concert/C extends the C language [kernighan88] to incorporate distributed computing abstractions. In Concert/C, a program in execution represents a process and communication end-points are abstracted into port data type objects through which messages are exchanged. Data are received through input ports (inports for short) and sent through output ports (outports for short). Outports are also known as bindings. The term port is used when it is not necessary to distinguish an inport from an outport. An inport consists of a queue of messages that arrived but were not processed yet. A binding consists of a pointer to (an address of) an inport. The type of a port is determined by the type of messages it can exchange. Concert/C provides operators for managing (creating and terminating) processes and for sending and retrieving messages through ports. The exchange of messages can be synchronous (abstracted as remote procedure calls [nelson81] [soares92]) or asynchronous (message passing). The design of Concert/C follows the Concert model [yemini89] [auerbach91] for distributed computing. In Appendix A, we describe the Concert model in detail and overview the Concert/C language.

The decision to prototype QuAL in Concert/C was highly motivated by the elegance with which distributed computation abstractions were added to C and by the implementation design of the language runtime system. Concert/C supports distributed computing by adding to C the concepts of processes and ports, and a concise set of operators. Concert/C is completely backwards compatible with ANSI C. Thus, for C programmers, the Concert/C learning overhead is reduced to understanding the few new concepts and operators added. The implementation design of the Concert/C runtime is anchored on sheltering heterogeneity at the transport and OS layers. Similarly, QuAL is designed to ease the development of applications expected to run over heterogeneous environments. Thus, the Concert/C runtime consists of a sound platform in which we can integrate the QuAL runtime.

In order to abstract QoS on communication streams, QuAL provides means to:

- Specify QoS measures and processing constraints of message streams.
- Access temporal properties (sending, arrival, and processing times) of messages and prioritize message processing accordingly.
- Specify control of QoS and processing of message streams via attached QoS and processing exception handler constructs.
- Automatically generate Management Information Bases (MIBs) that contain traces gathered from QoS and processing monitoring.

By handling communication QoS at the language level, QuAL *bridges heterogeneity at lower communication-service providers* (e.g., transport layer, OS). It is the language implementer's responsibility to translate the communication QoS demands specified into available services (e.g., transport-layer, session-layer, or application-layer services) that can best deliver the QoS requested. At compile time, QoS measures are translated to an internal representation known to QuAL's runtime. At connection-establishment time, QuAL's runtime chooses a communication-service provider for a connection based on its QoS demands and on the service providers available. The choice is performed during run time, since processes are created and interconnected dynamically and the underlying environment in which processes are instantiated are heterogeneous regarding the type of communication services offered and the standards followed.

By enabling the generation of MIBs, QuAL *promotes management that focus on application properties* rather than on the infrastructure of the underlying system. The MIBs generated contain information on the QoS demanded for application-level resources (e.g., a particular inter-process communication stream) and on the actual QoS that are being delivered. Thus, management entities have access to the configuration of applications and to their QoS statistics collected. Such information reflects the performance of the services being delivered by the underlying system in a per application-level resource basis and enables a finer-grained management that focus on improving application properties (e.g., reducing the average propagation delay for a communication stream). It is particular important in computing and communicating multimedia information, where each type of data imposes different constraints on the communication QoS. In such a diverse environment, management is hard to generalize and has to be performed in a per-communication basis. Furthermore, QuAL is a pioneer in viewing management as an integral part of application development. MIBs can be automatically generated by the runtime of QuAL by specifying a simple option during compilation.

One of the results of this research is the definition of the structure of MIBs for storing QoS statistics. The MIBs design constitute a first step in capturing QoS statistics into MIBs. The MIBs are defined according to the SMI [stallings93] standard and can be accessed by any management entity that follows the SNMP [stallings93] standard.

The following sections discuss these features in more details and present examples that illustrate the QuAL language constructs. Appendix B contains the description of the complete syntax for QuAL constructs. When showing an example or defining the syntax, the following convention is used. Keywords and constructs in QuAL are written in **bold** face, in Concert/C are <u>underlined</u>, and in C are plain text.

2.1 Specification of QoS Measures and Processing Constraints

QuAL provides constructs for the specification of QoS measures that define network-level (e.g., propagation delay) and application-level (e.g., delivery rate) properties that streams being communicated should satisfy. QoS measures are specified in a per-port basis. The port data type is extended to include the specification of QoS attributes (that are dynamic parameterizable components of a port's type). The binding mechanism is extended to guarantee that only ports with *compatible* QoS attributes are bound. The concept of compatibility will be further elaborated in the following sections when the QoS attributes are defined.

QoS measures are used by QuAL's runtime to reserve communication and processing resources and to detect periods in which the required quality is violated. The runtime translates these abstract QoS measures (e.g., propagation delay) into transport-specific service access point parameters (e.g., processing and buffering resources that need to be allocated at each nodes in the path from the origin to the target to guarantee a bounded propagation delay). Furthermore, the runtime also monitors the services being delivered by the underlying environment to detect QoS violations (e.g., it stamps each message with its sending and arrival times to calculate propagation delay).

Real-time language constructs are added to provide means to express execution timing constraints (e.g., processing start time and deadline, hard or soft deadlines, etc.) necessary to specify many QoS measures. Real-time applications impose critical timing constraints. Processing resources are typically finely allocated when running real-time applications. QuAL includes real-time constructs.

In Sections 2.1.1 and 2.1.2, we describe in details the specification of network-level and application-level QoS measures in QuAL, respectively, and in Section 2.1.3 we describe the real-time language constructs in QuAL.

2.1.1 Specification of Network-level QoS Measures

Network-level QoS measures are specified using the following set of abstract *QoS attributes*: loss tolerance, permutation tolerance, maximum end-to-end delay, maximum inter-message delay, average transmission rate, peak transmission rate, and recovery time. Application developers specify through these attributes the QoS measures of importance for a communication and intervals in which these measures' values must be contained. QuAL monitors QoS based on the QoS attributes specification as follows. At compile time, each QoS attribute specification is mapped into instances of qos, qos^*, and B, according to the framework for QoS monitoring defined in Section 1. The instances are chosen to capture the QoS measures and value intervals specified. During run time, these functions are evaluated for each stream s_i communicated and QuAL's runtime raises an exception whenever $qos(s_i) - qos^*(s_i) > B(s_i)$. An exception signals that a specific QoS measure has a value not contained in the interval defined. The exception handling mechanism is further explored in Section 2.3.

Example 2.1 illustrates the specification of QoS measures in QuAL. The declaration of inports and bindings in QuAL consists of the attachment of QoS measures specification (defined inside the brackets following the keyword **`realtm`**, short for real-time) to a port specification in Concert/C (that consists of the keyword `receiveport` followed by brackets surrounding the specification of the type of the messages exchanged through the port and by the name of the object being declared). The example shows the declaration of the inport `video_input` that receives video frames of type `video_frame_t` and of the binding `video_output` that sends video frames of the same type. The * symbol distinguishes the declaration of a binding from the declaration of an inport. It represents a reference to (an address of) an inport. The process receiving video frames is willing to tolerate a probabilistic loss of up to 10^{-6} (specified by the term **`loss`** 6) and an arbitrary permutation (**`permt;`**) during the communication. It is capable of processing up to 15 messages/sec in average (**`rate ... - sec`** `15;`)[1] and it supports a peak rate of up to 20 messages/sec (**`peak - sec`** `20;`). It also requires the sender to deliver at least 10 messages/sec in average (**`rate sec`** `10 - ...;`). The transmission delay should be no higher than 35 ms (**`delay ms`** `35;`) and the inter-message delay no higher than 1/3.0 of a second (**`inter-delay sec`** `1.0/30;`). Any recovery should take no longer than 3 ms (**`recovery ms`** `3;`). The process sending the video frames, however, is capable of sending an average of 25 messages/sec (specified by **`rate - sec`** `25` in the declaration of `video_output`) through `video_output`, with temporary peaks of 30 messages/sec (**`peak - sec`** `30;`). It has no restriction regarding loss and permutation at the communication level (specified by the **`NULL`** keyword after the **`loss`** and **`permt`** keywords). The loss and permutation QoS measures can be further restricted if this port is bound to an inport with more restrict QoS measures. The sending process tolerates a maximum time period of 4 ms for recovery from any connection problem on `video_output` (**`recovery ms`** `4;`).

Example 2.1: Specifying Network-level QoS Measures.

```
% The Receiver
realtm {loss 6; permt;
        rate sec 10 - sec 15; peak - sec 20;
        delay ms 35; inter-delay sec 1.0/30.0;
```

---

[1]The symbol - is used to separate the lower from the upper limits of the interval being specified. When the lower or the upper bound are not specified a value of 0 or inifinite is assumed, respectively.

```
        recovery  ms  3; }
receiveport {video_frame_t} video_input;

% The Sender
realtm  {loss  NULL;  permt  NULL;
        rate - sec 25; peak - sec 30;
        recovery  ms  4; }
receiveport {video_frame_t} *video_output;
```

The QoS measures specified in the example above define an *interval of acceptance* in which the QoS delivered by the underlying system to the application must be contained. The notion of interval of acceptance is derived from the QoS framework defined in Section 1. Each attribute specification is mapped into a qos function that will be measured during communication. The QoS measures' values define qos^* (optimal QoS function) and B (maximum deviation bound function) constant functions. For each stream s_i communicated, QuAL's runtime raises an exception whenever $qos(s_i) - qos^*(s_i) > B(s_i)$. Thus, optimal and maximum deviation bound values define intervals within which the QoS values delivered to the application must be contained, or a violation occurs. In Example 2.1, the QoS measures specification of `video_input` can be interpreted in the following way: (1) an optimal probabilistic loss of 0 and a deviation bound of 10^{-6} define the interval from 10^{-6} to 0, (2) an optimal permutation rate of 0 and an infinite deviation bound define the infinite interval, (3) an optimal rate of 15 messages/sec and a bound of 0 define the interval from 0 to 15, (4) an optimal null transmission delay with a bound of 35ms define the interval from 0 to 35, etc.

Computer and communication resources are allocated using transport-layer and operating-system services in order to establish a connection with QoS complying with the requested measures. QuAL's implementer is responsible for mapping these abstracts QoS measures (e.g., transmission rate) into lower-level service requests (e.g., allocation of specific amounts of bandwidth and buffering resources at all nodes involved in the communication) to establish a connection that best delivers the QoS demanded. The choice of the underlying communication mechanism to be used (e.g., transport-layer ST-II [RFC1190] connection, an application-layer MCAM [keller93] connection, etc.) is performed dynamically, when the communicating ports are bound. This is because only at binding-time the runtime knows the configuration of the communication (e.g., same machine, across machines over a local area network, or over a wide area network) and of the environment in which the communicating processes are (e.g., support for ST-II, support for MCAM, etc.).

Both sender and receiver processes participate in the communication QoS measures negotiation (symmetric negotiation), as opposed to existing mechanisms where either the sender or the receiver chooses the QoS of the communication while the other needs to comply with the choice (asymmetric negotiation) (e.g., MCAM [keller93]). The binding mechanism is extended to guarantee that only ports with compatible QoS attributes are bound. Two ports have compatible network-level QoS attributes if the intersection of their acceptance intervals is non-null for each network-level QoS attribute specified. In Example 2.1, the ports have compatible attributes and the intersection intervals define a rate that can range from 10 to 15 messages/sec, with a peak rate no higher than 20 messages/sec, a recovery time no longer than 3 ms, a loss no higher than 10^{-6}, and an arbitrary permutation rate. While bound, the type of the network-level QoS attributes of the ports bound must be comprised in the intersection set. Bound ports can only communicate if the runtime is capable of establishing a connection whose QoS values belong to the respective intersection intervals.

QuAL is unique in providing a strongly-typed language-level abstraction for QoS measure specification that shelters heterogeneity at lower layers (e.g., transport-layer and OS). The language-level abstraction maps symmetric QoS negotiation protocol and QoS monitoring into the simple concept of run-time type-checking. QuAL exposes the application developer to a single abstraction for the specification of QoS, independent of the underlying system service provider. QuAL promotes code portability and reusability over heterogeneous environments by bridging the gap between language-level abstractions and underlying system services.

2.1.2 Application-level QoS Measures

Application-level QoS measures specify application-dependent constraints that must be imposed on the streams being communicated. For instance, if a sender is sending more messages than a receiver is capable of processing, the communicating processes can agree under which circumstances (e.g., sending time or current system load) a message sent should be delivered to the target for processing or be discarded. Another example is when processes dictate bounds for the arrival time of messages and want to discard messages considered to be late. The main goal is to instrument applications with a mechanism to negotiate, restrict, and control the access to their communication end-points. Control of QoS measures provides protection similar to the one provided by a fuse when installed in an electrical circuit, that is, it preserves applications from damaging whole systems when unpredictable logical glitches happen.

Application-level QoS measures can represent application-dependent constraints on a single message stream (e.g., application-level rate control) or inter-dependencies among multiple streams (e.g., synchronization). Single-stream QoS measures define constraints that are related to a particular stream. Inter-dependent QoS measures define QoS measures of a particular stream that depend on QoS properties of a group of streams. Inter-dependent QoS measures are also known as group QoS measures.

Contract identifications are used to recognize application-level QoS measures. A contract for a given port specifies that the expression $\Delta qos'(s_i) \leq \beta'(s_i)$ (as defined in Section 1) must be evaluated (or monitored) for every stream s_i communicated through the port. qos' represents an application-dependent QoS function and β' its deviation bound function. For input ports, this contract is enforced on arriving messages, whereas for bindings it is enforced on messages as they are sent. In the QuAL model of computation, messages that violate a contract are either discarded or forwarded to exception handler ports, depending on options specified in the definition of the port. The exception handling mechanism is further explained in Section 2.3.

Contracts are part of port types, and can be divided into two classes: (1) the ones that the port complies with and (2) the ones that any other port must comply with in order to be bound to the port being declared.

Example 2.2 illustrates the specification of single-stream application-level QoS measures in QuAL. The outport `video_out` is able to comply with the list of contracts inside the brackets following the keyword **`cmpl`**: `rate_20`, `rate_15`, or to no contract at all (the **`NULL`** value). What these contracts actually represent will be better understood later, when we explain how to enforce and to implement them. Contracts are application-dependent and have meaning only in the context of a particular port. In the example, the identifiers `rate_20` and `rate_15` represent the contracts that guarantee that the rate will be no higher than 20 messages/sec and 15 messages/sec, respectively. The example also illustrates the declaration of the inport `video_in` that can only be bound to an outport that complies with the contract listed inside the brackets following the keyword **`bnd_cmpl`**: `rate_15`.

Example 2.2: Specifying Application-level QoS Constraints: Controlling the Rate of Messages

```
% The Sender
realtm {cmpl {rate_20;  rate_15;  NULL;};}
receiveport {video_mss_t} *video_out;
...
% The Receiver
realtm  {bnd_cmpl  {rate_15;};}
receiveport {video_mss_t} video_in;
```

Example 2.3 illustrates the specification of inter-dependencies among multiple streams in QuAL. On the sender side, the bindings `video_outport` and `audio_outport` are declared. The bindings declared are capable of complying with the contract specified inside the brackets following the keyword **`grp_cmpl`**: `audio_video`. The keyword **`grp_cmpl`** classifies the contract as representing inter-port constraints instead of single-stream constraints. The contract `audio_video` constraints audio messages to be synchronized with video messages. The contract identification for inter-port constraints is an enumeration identifier. Each element in the enumeration represents one of the ports that are inter-related. When a port is defined as being capable of complying with an inter-port QoS, the declaration must identify which element of the enumeration that port represents. In the example, the enumeration `audio_video` contains the elements `audio` and `video`. The binding `video_outport` represents the `video` element. The representation is specified by the `video` word inside the brackets following the word `audio_video` in its declaration. Analogously, the binding `audio_outport` represents the `audio` element. On the receiver side, the inports `video_inport` and `audio_inport` can only be bound to ports that can comply with one of the contracts in the brackets following the keyword **`grp_bnd_cmpl`**. In the example, only the contract `audio_video` is specified. `video_inport` must be bound to the port representing the `video` and `audio_inport` must be bound to the port representing the `audio`.

Example 2.3: Specifying Inter-port QoS Constraints: Audio and Video Synchronization

```
% The Header File
enum {audio, video} audio_video;
% The Sender
realtm {grp_cmpl {audio_video[video];};}
receiveport {video_mss_t} *video_outport;
...
realtm {grp_cmpl {audio_video[audio];};}
receiveport {audio_mss_t} *audio_outport;
...
% The Receiver
realtm {grp_bnd_cmpl {audio_video[video];};}
receiveport {video_mss_t} video_inport;
realtm {grp_bnd_cmpl {audio_video[audio];};}
receiveport {audio_mss_t} audio_inport;
```

Two ports have compatible types with respect to the application-level QoS attributes if they are able to comply with each other's contract demands. More specifically, they must be *compatible at the sending and at the arrival end*. Two ports are compatible at the sending end if there is a non-null intersection for each of the following sets: (1) the set of contracts that the outport is capable of complying with (specified after the keyword **`cmpl`** in the outport definition) and the set of contracts that the inport demands from a binding (specified after the keyword **`bnd_cmpl`** in the inport definition), and (2) the set of inter-port contracts that the outport is capable of complying with (specified after the keyword **`grp_cmpl`** in the outport definition) and the set of inter-port contracts that the inport demands from a binding (specified after the keyword **`grp_bnd_cmpl`** in the inport definition). Similarly, two ports are compatible at the receiving side if there is a non-null intersection between the QoS specification of the inport, and the QoS that the outport demands from an input port. In Example 2.2, the two ports are compatible since the inport `video_in` requires that an outport complies with the contract `rate_15`, and the outport `video_out` is capable of complying to it. Similarly, in Example 2.3, the outport `video_outport` is compatible with the inport `video_inport`, and the outport `audio_outport` is compatible with the inport `audio_inport`.

An application developer must bind to each contract a *monitoring function* that checks for contract compliance. Monitoring functions have no access to the contents of a message, only to its temporal properties (sending and arrival times) and to an index that identifies the order of the message in the stream being communicated. A monitoring function can be any function written in a subset of C that allows for the worst case performance analysis of its execution time. Such property provides the QuAL runtime with enough information for allocating processing resources to guarantee the real-time execution of the function. Monitoring functions for outport contracts are called every time a message is sent whereas monitoring functions for inport contracts are called every time a message arrives. Each monitoring function is executed in the scope of a separate process created by the runtime when a connection is established. Monitoring functions return a true value if the temporal properties of the message comply with contract and a false value otherwise. Messages that do not comply with the contract are either discarded or forwarded to a designated exception handler port, depending on options defined in the port specification. The exception handling mechanism is further explained in Section 2.3.

Example 2.4 illustrates how to define and designate monitoring functions to implement the contracts specified in Example 2.2. The example shows the declaration of two functions: `reduce_30_15` and `reduce_30_20`. `reduce_30_15` returns true if the integer passed as first argument is odd, and false otherwise. `reduce_30_20` returns true if the integer passed as first argument is a multiple of 3, and false otherwise. These functions can be used as monitoring function for outport contracts, where the first argument represents the message index and the second argument its sending time. In Appendix B, we explain in more details constraints and assumptions that QuAL imposes on the signature of functions used as monitoring functions. In this example, `reduce_30_15` establishes that messages with an odd index comply with the contract, whereas the others do not. This is a straight forward way of reducing the rate by half. Similarly, `reduce_30_20` reduces the rate by two thirds. In Example 2.4, these functions are bound to the contracts `rate_15` and `rate_20`, respectively, of the port `video_in`, through the QuAL operator **`assg`**. This example illustrates the diversity of monitoring that can be performed using this scheme, allowing the same outport to be connected to different inports with different constraints. The same `video_out` port can be used to send a video sequence over lines of different bandwidth and to receivers with different processing capabilities. It is up to the receiver to select the contract that the outport must comply for a specific communication and the compliance to the contract will tune (reduce) the rate of messages accordingly.

Example 2.4: Implementing Contracts: Controlling the Rate of Messages

```
% The Sender
realtm {cmpl {rate_20; rate_15; NULL;};}
receiveport {video_mss_t} *video_port;
```

```
...
int reduce_30_15(int index, time_tp time) {
   % Return 0 if index is even and 1, otherwise.
}
int reduce_30_20(int index, time_tp time) {
   % Return 0 if index is not multiple of 3 and 1, otherwise.
}
...
assg(video_port, rate_15, reduce_30_15);
assg(video_port, rate_20, reduce_30_20);
```

Example 2.5 extends Example 2.3 to illustrate how the audio and video ports can be synchronized. The function `sync` is defined as taking three arguments. The first argument identifies the port from which the message is coming from, the second argument represents the message index, and the third argument its sending time. If a message is sent through the `audio_outport` and the message before this one was also originated from `audio_outport`, the function returns a false value. Similarly, if the current and previous messages are originated from the `video_outport`, the function also returns a false value. The function `sync` is then designated as the monitoring function for the contract `audio_video`, through the operator **`assg_grp`**. This monitoring guarantees that a video message is always sent after an audio message and vice versa. Messages for which the monitoring function returns a false value are discarded or forwarded to an exception handler port, depending on additional options as it will be explained in Section 2.3. The processes handling the exception handler ports will be responsible for managing the messages that did not go through, possibly trying to send them again, after a certain interval of time.

Example 2.5: Synchronizing Audio and Video

```
% The Header File
enum {audio, video} audio_video;
% The Sender
realtm {grp_cmpl {audio_video[video];};}
receiveport {video_mss_t} *video_outport;
realtm {grp_cmpl {audio_video[audio];};}
receiveport {audio_mss_t} *audio_outport;
...
int sync(int origin, int index, time_tp time) {
   % If origin is audio and last message sent was audio, then return 0 (delay audio stream)
   % If origin is video and last message was video, then return 0 (delay video stream)
```

```
    % Otherwise, return 1.
}
...
assg_grp(audio_video, sync);
```

QuAL is unique in defining constructs for the specification of customized application-level constraints on communicating streams that allows fine tuning of the run-time allocation and monitoring mechanism to specific application requirements.

2.1.3 Specification of Processing Constraints

In order to be able to specify certain QoS measures (e.g., deadlines to process messages), real-time language constructs are added to the base language to enable the expression of execution timing constraints. QuAL supports the notion of a *real-time block*. A real-time block has a *frame* associated with it, that corresponds to the minimum frequency (or period) of the occurrence of the phenomena or activity handled by the block. This activity is called *task*. A real-time block can be activated at specific times, or in response to an external event, such as the arrival of a message. Once activated, a block must complete its task according to certain timing constraints, such as before a certain deadline or at the end of the current frame, whichever happens first. Furthermore, once activated, a block cannot be reactivated before the end of the current frame.

Timing constraints can be either *hard* or *soft*. Hard constraints are the ones that must be met or else the application will deliver unacceptable results. Soft constraints indicate ideal targets which should be met when the system is not overloaded, but that can be missed occasionally.

The real-time execution mode can be of three sorts: *periodic*, *sporadic*, or *aperiodic* . Periodic tasks are those which have to be processed at regular intervals, and must be completed before the next is due. Sporadic tasks are asynchronous tasks that may have hard deadlines, but for which there is a minimum inter-arrival duration between instances. Aperiodic tasks are sporadic tasks that have only soft deadlines and for which no minimum inter-arrival time interval is known.

To ensure that a program is executed in a predictable way, its *computational cost* is calculated and a *scheduleability analysis* of its processes is performed. The computational cost is the maximum amount of processor time required to execute the sequential program of a task to completion on a dedicated uniprocessor. A scheduleability analysis verifies if there is a sequential scheduling of the processes executing in the machine such that none of them violates its timing constraints. The calculation of the computational cost makes possible an analysis of the worst-case system load and of processes scheduleability. Through these analyzes, hard timing constraints are guaranteed not to be missed and resources are managed to prevent a system from reaching an overloaded state.

QuAL requires the specification of a time-out period for every instruction or sequence of instructions whose execution period cannot be calculated in advance (at compile time). Examples of this type of instructions are repetition loops, recursive function calls, and blocking instructions (e.g., reading or writing from a device).

QuAL's support for soft execution permits to trade predictability for higher throughput. A worst case performance analysis is performed before a hard real-time task can be accepted for execution. On the other hand, soft real-time tasks begin executing, even if it is known that timing constraints may not be complied with during certain periods.

Example 2.6 illustrates how video can be processed in real-time in the context of a multimedia conferencing application. In this context, it is desirable to give priority to audio display than to video display, since our audition is more sensitive to high jitters and data loss than our vision. Therefore, the video display can execute in soft mode, whereas the audio display would probably execute in hard real-time mode. The keyword **`within`** identifies the real-time block and the semantics are as follows. If there are not enough processing resources available to allow the execution of the real-time block complying with its timing constraints, control is passed to the statement following the **`until`** condition. Otherwise, the process enters the real-time mode of execution. The **`do`** block represents the task that must be performed. This task will be executed according to the timing constraints specified inside the parentheses following the keyword **`within`**, until the **`until`** condition evaluates to a true value. The timing constraints specify a sporadic soft mode of execution (the **`soft;`** clause), with a maximum of 30 activations per second (**`period sec`** `30;`). The task will execute for the first time only after the expression following the keyword **`after`** (`select(video_inport)`) is satisfied, that is, there is a message in the input port `video_inport`. Similarly, every execution is triggered by the satisfaction of the expression following the keyword **`atEvent`** (`select(video_inport)`), that also demands the arrival of a message at the same port. Since the display involves I/O and it is not known how long it will take, a time-out expression is required. So, either the display is executed in 60 seconds (**timeout`(sec`** `1/60.0`**`)`**), or an exception is raised. An exception can also be raised if the timing constraints of the process cannot be met. That is, if video messages arrive but the process cannot be scheduled in intervals of 30 seconds. The exception handling mechanism is described in Section 2.3.

Example 2.6: Handling Video in Real-time.

```
main()
{  ...
   within (soft; sporadic; atEvent(select(video_inport));
            period sec 30; after(select(video_inport));)
   do { timeout(sec 1/60.0) {
         /* display video frames */ }}
```

```
    until (false);
}
```

2.2 Real-time Ports

QuAL defines real-time ports, that is, ports that allow the access to temporal properties (sending, arrival, and processing times) of messages crossing them. The support for real-time ports promotes the monitoring and study of communication temporal properties. QuAL provides a set of operators to access these temporal properties. The operators are simple extensions of the Concert/C operators for receiving and processing messages.

Example 2.7 illustrates the retrieval of temporal properties in QuAL. The call to **`receive_tm`**, similarly to a call to the Concert/C operator `receive`, causes a message to be dequeued from the port p and its contents to be stored in the memory position designated by `&m`. Nevertheless, **`receive_tm`** also causes the message sending, arrival, and processing times to be stored in the memory positions designated by `&sendtm`, `&arrvtm`, and `&proctm`, respectively. The first argument of the **`accept_tm`** operator, similarly to the Concert/C operator **`accept`**, is of type *function port*. Objects of type function port are both a C function and a Concert/C input port. Because they are of type function, they can be called as any normal function. Because they are of type input port, they contain a queue of messages that represent remote calls to the function. In Example 2.7, the call to **`accept_tm`** causes a message (representing a remote procedure call) to be dequeued from the function port `f` and a call to `f` to be executed. The call arguments are the contents of the message dequeued. However, before the call to `f` is made, the message temporal properties are stored in the memory positions designated by `&sendtm`, `&arrvtm`, and `&proctm`.

Example 2.7: Accessing Communication Temporal Properties in QuAL.

```
...
receive_tm(p, &m, &sendtm, &arrvtm, &proctm);
accept_tm(f, &sendtm, &arrvtm, &proctm);
```

In QuAL, message processing may be prioritized based on: (1) order of arrival (as in Concert/C) or (2) sending time. QuAL introduces extended versions of the Concert/C operators `select` and `poll` that indicate which port in a set of ports has a message that was sent within a certain interval of time. QuAL also extends the Concert/C operators `receive` and `accept` and its own operators **`receive_tm`** and **`accept_tm`** to prioritize message dequeueing based on the message sending time.

Example 2.8 illustrates prioritization of message dequeueing in QuAL. The call to **`rtm_receive`** dequeues from port `p` a message sent in the interval between the time stored in `t` and now. The current time is the result of the call to the function `time`. The call to **`rtm_receive`** also saves the contents of the message in the memory position designated by `&m`. The call to **`rtm_receive_tm`** works similarly to the call to **`rtm_receive`**, but it also causes the message's temporal properties to be saved in the memory positions provided. Analogously, the operators **`rtm_accept`** and **`rtm_accept_tm`** extend the operators `accept` and **`accept_tm`** to prioritize the dequeueing of messages based on the message sending time. These operators only dequeue messages sent within the time interval delimited by their first two arguments.

Example 2.8: Prioritizing Message Service in QuAL

```
...
rtm_receive(t, time(), p, &m);
rtm_receive_tm(t, time(), p, &m, &sendtm, &arrvtm, &proctm);
rtm_accept(t, time(), f);
rtm_accept_tm(t, time(), f, &sendtm, &arrvtm, &proctm);
```

Existing language-level communication abstractions do not capture communication temporal properties. Thus, application developers must incorporate the sending time in the contents of the message, and processes must trust each other that the time is correct. The access to the arrival time is more complicated. The receiving process must block waiting for the message and check the time soon after a signal is received. Since most environments allow for concurrent execution of processes, this time may not represent the exact time in which the message was received, but instead the time in which the receiving process was scheduled for execution. In QuAL, the manipulation of temporal properties is transparent to application developers and resides at a lower layer (QuAL's runtime), similarly to the way that network-layer header processing is transparent to the transport layer. Furthermore, time is checked atomicaly with the message sending and arrival processing.

The communication abstraction in QuAL is completely backwards compatible with the communication abstraction in Concert/C. Thus, all Concert/C operators can be used to access real-time ports. In this case, the temporal properties are simply discarded.

2.3 Specification of QoS and Processing Control

When requested QoS or processing timing constraints are violated, QuAL's exception handling mechanism is responsible for calling upon proper corrective actions. The exception handler will control the exceptional conditions and gracefully recover from these transient periods of degradation.

If either an application or the underlying system violates the QoS constraints of a communication stream, an exception message is sent to a user-defined exception handler port associated with the connection. QuAL run-time QoS monitoring prevents applications from misusing the resources allocated for a communication (e.g., sending data in a rate higher than the connection is capable of delivering) and monitors connections to guarantee that the lower layers are delivering the QoS negotiated during connection establishment time (e.g., the propagation delay is not higher than the delay negotiated). Exception handler processes are bound to the exception handler ports and take the proper user-defined corrective measures whenever messages arrive at the exception handler ports.

A general overview of the QoS control abstraction in QuAL is depicted in Figure 2.1. The abstraction consists of processes (represented by big rectangles), data inports (represented by squares), bindings (represented by circles), and exception handler ports (represented by small black rectangles). Exception handler ports need not belong to the same processes that own the bindings or the inports being monitored. The dashed line in the figure signifies a virtual partition of the application-level processes. The components are depicted in their respective layers which are separated using a thick horizontal line. The arrows represent information flow between components. From the application-developer's point of view, a communication is abstracted as in Concert/C, that is, using the notion of a port and a binding that are used to communicate the application's data. The application's data flow is represented by the dashed arrow at the application layer. In reality, the data is sent to runtime components that communicate the data as well as control information to the other end. QuAL adds to this abstraction the notion of a set of exception handler ports that receive a message when QoS violations occur. These messages are sent by Communication Monitoring Processes (CMPs) implemented by QuAL's runtime to monitor connections. CMPs are accessible at the application-layer to enable applications to customize them for monitoring application-dependent QoS.

Example 2.9 illustrates the specification of exception handler ports in QuAL. The port `video_input` is defined as having network-level and application-level QoS constraints. If any network-level QoS violation occurs (e.g., the propagation delay is higher than 35 ms, the average loss rate is higher than 10^{-6}, the process is sending more than 15 messages per second, etc.), a message is sent to the port `manage_conn`. Furthermore, any message that violates the single-stream application-level constraint chosen for the connection (`rate_20 or rate_15`, depending on which one was chosen by the port's binding) is forwarded to the inport `trash`. Similarly, any message that violates the inter-port constraint `audio_video` is forwarded to the port `video_delay`. The declaration of exception handler ports for network-level, single-stream application-level, and inter-port application-level QoS constraint violations follow the keywords **`net_handler`**, **`app_handler`**, and **`grp_app_handler`**, respectively, in the **`handlers`** clause of a port definition.

Example 2.9: Specifying Handlers for QoS Control

```
realtm {loss 6; permt;
        rate sec 10 - sec 15; peak - sec 20;
        delay ms 35; inter-delay sec 1.0/30.0;
```

```
        recovery ms 3;
        cmpl {rate_20; rate_15;};
        grp_cmpl {audio_video[video];};}
handlers {net_handler manage_conn;
          app_handler trash;
          grp_app_handler video_delay;}
```
receiveport {video_frame_t} video_input;

To deal with periods in which the required QoS are being violated, QoS parameter re-negotiation is possible during execution time. A re-negotiation can happen at anytime during process execution and it will occur concurrently with the sending and arrival of messages. QuAL provides two operators to ease QoS re-negotiation. The operator **`qos_get`** allows the retrieval of the current connection QoS values of a port and the operator **`re_negotiate`** causes the re-negotiation to start.

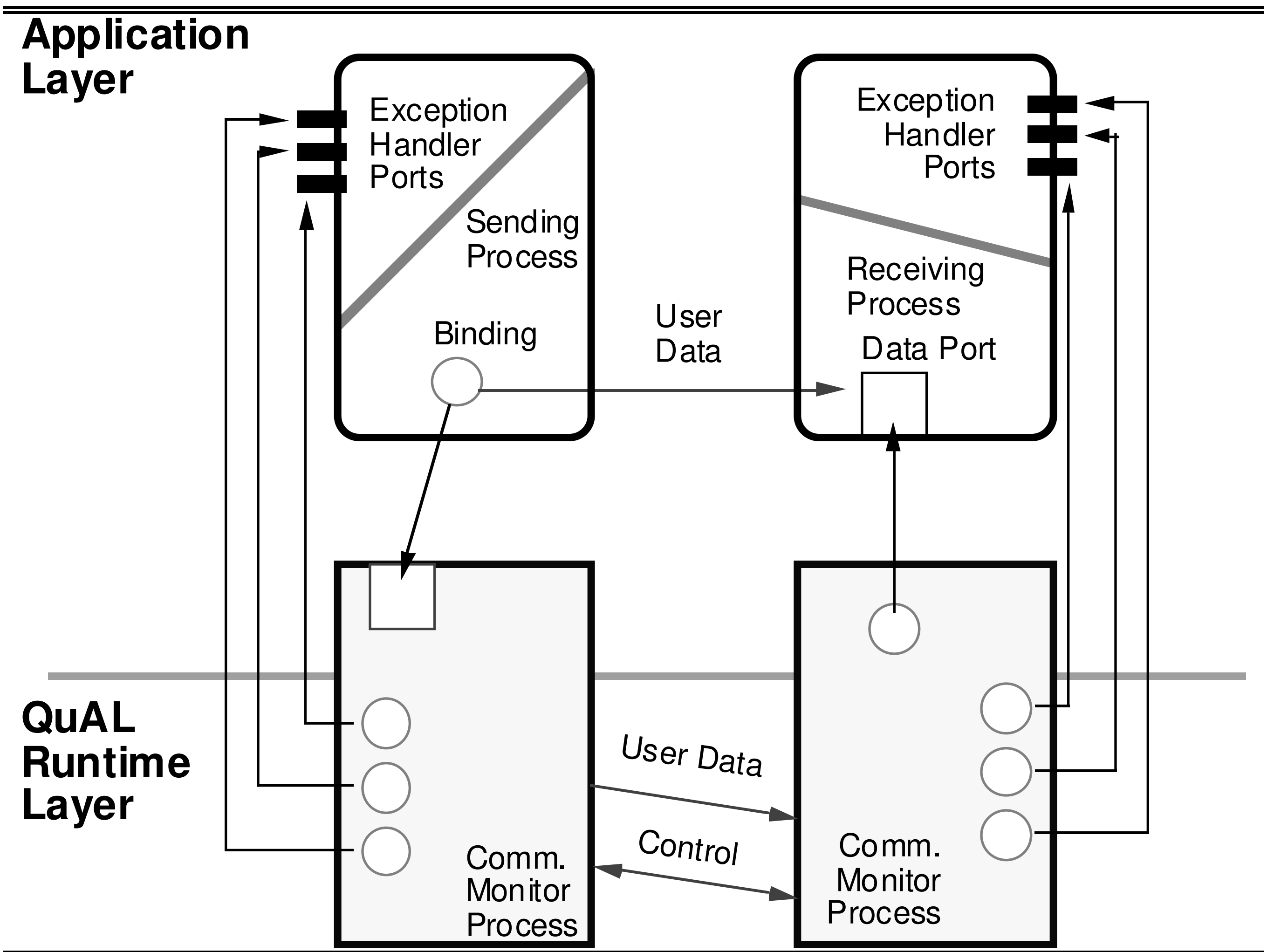

Figure 2.1: Communication Abstraction in QuAL

Example 2.10 illustrates QoS re-negotiation in QuAL. Firstly, the **qos_get** operator is used to access the delay negotiated for the current connection for port `p`. The value of the delay is returned in the memory position designated by &`delay`. Secondly, the new delay is calculated. In this case, it is the current value minus 50 ms. Finally the re-negotiation is requested through the **re_negotiate** operator.

Example 2.10: QoS Re-negotiation in QuAL

```
qos_get  (p)
{ delay sec &delay;
}
delay -= (ms) 50;
re_negotiate  (p)
{ delay sec delay;
}
```

Exception handlers can also be defined for soft timing constraint violations. QuAL recognizes three types of exceptions: (1) the time-out period for a block of instructions expires and the execution of the block does not reach its end; (2) the deadline for the execution of a task is reached and the execution of the task is not over, and (3) a task requires some responsiveness in its execution, that is, the task execution has to start within a certain interval, and this is not achieved.

Example 2.11 illustrates how exception handlers for processing constraint violations control are defined in QuAL. If the time-out of 1/60.0 of a second expires, the block of code following the keyword **expired** is executed. In this example, it sets the variable `final` to TRUE causing the **until** condition to become TRUE and the real-time execution to end. The timing constraints have the additional requirement that the execution of the block should start no later than 1/30.0 of a second after the **after** condition becomes TRUE, specified by the **response sec** `1/30;` term. If the responsiveness constraint is not satisfied, the block of code following the keyword **responsiveness** is executed. In this case, it discards the video frame, since there will be no time to display it properly. Similarly, if the execution does not finish after 1/30.0 of a second the block of code following the keyword **deadline** is executed. The deadline is assumed to be 1/30 of a second because the timing constraints specify that the block must execute 30 times per second (**period sec** `30;`).

Example 2.11: Specifying Handlers for Processing Constraints Control

```
within (soft; sporadic; atEvent(select(video_inport));
        period sec 30; after(select(video_inport));
        response sec 1/30.0;)
do { timeout (sec 1/60.0)
```

```
        { /* display video frames */ }
        expired {final = TRUE;};}
responsiveness { /* discard the video frame */ }
deadline { /* discard next video frame */ }
until (final);
```

The exception handling mechanism for constraint violations enables the implementation of graceful recovery and degradation.

2.4 Application Management Automation

The data on the communication QoS provision used to monitor user-defined measures is also used by QuAL's runtime to generate and update application-customized Management Information Bases (MIBs) that contain traces gathered from QoS and processing monitoring. External management entities access the MIBs generated to control the QoS provided by the underlying system (e.g., re-configuring the network to reduce the propagation delay). Applications, on the other hand, use the MIBs to control their own performance according to the services being delivered (e.g., tuning message processing rate based on the number of buffered messages).

QuAL's runtime follows the *Simple Network Management Protocol* (SNMP) [stallings93] standards for the generation of MIBs. The term SNMP is actually used to refer to a collection of specifications for network management that includes the protocol itself, the definition of a database, and associated concepts. SNMP was designed to be an application-level protocol that is part of the TCP/IP [stevens90] protocol suite. The model of network management that is used for TCP/IP network management encompasses the following elements:

- *Management station*: that serves as the interface for the human network manager into the network management system. Usually, it has a set of applications for data analysis, an interface by which the manager may monitor and control the network, and a database of information extracted from the MIBs of all managed entities.
- *Management agent*: that responds to requests for information and requests for actions from the management station, and may asynchronously provide the management station important but unsolicited information.
- *Management Information Base (MIB)*: that is a collection of *objects*. Each object is essentially a data variable that represents one aspect of the managed agent. The MIB functions as a collection of access points at the agent for the management station. A management station performs the monitoring function by retrieving the value of MIB objects.

- *Network-management protocol*: that controls the communication between the management station and the agents. The protocol used is the SNMP protocol, which includes the following capabilities:
    - Get: enables the management station to retrieve the value of objects at the agent,
    - Set: enables the management station to set the value of objects at the agent,
    - Trap: enables an agent to notify the management station of significant events.

QuAL generates management information for every program module that is compiled with the management option. QuAL's runtime implements SNMP agents that store this information in MIBs defined according to the *Structure of Management Information* (SMI) [stallings93]. SMI specifies the general framework within which an SNMP MIB can be defined and constructed.

QuAL generated MIBs are customized in that they reflect peculiarities of the applications and allocated resources. The QuAL runtime maintains three types of MIBs: *application MIB, inport MIB* and *outport MIB*. The ASN.1 [rose91] definition of these MIBs according to the SMI standard can be found in [florissi94]. The application MIB has one row for each managed QuAL application running on a system. The inport MIB and the outport MIB have one row for each QoS-dependable inport or outport connection, respectively, belonging to applications defined in the application MIB. An entry in the application MIB describes the current status of the application and statistics on its distributed activities and real-time executions. It contains information such as the time the application started and its current status (e.g., interrupted, executing, executing in real-time, etc.), the number of QoS-dependable and non QoS-dependable inports and outports exported or imported, respectively, by the application, the number of QoS-dependable inports and outports connected, the number of rejections to QoS-dependable inport connection establishment requests, the time when the last rejection occurred, the number of times the process executed in real-time mode, the number of rejections to real-time execution requests, the time when the last rejection occurred, the number of exception messages received by the application regarding problems at the communication level or real-time execution, etc. An entry in the outport MIB describes QoS-dependable outport connection characteristics and contains statistics on its traffic performance. It contains information such as identification of the inport connected to this one, the connection QoS parameters negotiated with the network service provider, the time in which the connection was established, the number of messages sent, the volume of data sent measured in kilo bytes, the accumulated inter-message generation delay, the number of times there was a problem in the connection, the time when the last connection problem occurred, the number of QoS violations on this connection, etc. The inport MIB contains similar information for QoS-dependable inport connections. In Section 3.1, we discuss the design of the MIBs in greater details.

External (outside the application domain) network-management entities access these MIBs using the SNMP protocol to perform monitoring. Management entities are exposed to the configuration of the applications running on the system and can use these data to customize the monitoring and control of service-provider resources (e.g., network-level connections) accordingly. Based on the configuration of the applications running and on the particular QoS requirements of their connections, recovery from a broken link, for instance, can be geared towards isolating and re-configuring the most demanding connections first.

Applications can also monitor the performance of the system and of the applications themselves by accessing the MIBs. QuAL recognizes, in all managed modules, the SNMP get operation that enables the retrieval of the value of objects in the MIBs. These access operations are performed through the QuAL runtime SNMP agent. Applications see the MIBs as an automatic way of maintaining management information and use them to control their performance. By supporting the SNMP get operations, management applications can be directly implemented in QuAL.

QuAL also supports the specification of *traps*, a mechanism to asynchronously notify QuAL applications of exceptional conditions recorded in the MIBs without polling them. These conditions are reported to the application in the form of trap messages that are sent to designated exception handler ports. The trap mechanism is similar to the exception signaling mechanism employed by QuAL's runtime when a communication QoS violation occurs, but of a more general purpose. In the trap mechanism the exceptional condition is based on any of the statistics collected in the MIBs, that contain information on the applications themselves and on all the communications. Thus, QuAL eases the development of applications to manage their own applications and to tune their performance based on the MIBs' contents.

A playout time control of a multimedia conference application illustrates how management information collection automation and the trap mechanism in QuAL can ease the management of real-time applications with QoS-demanding communications. A conferencing system in QuAL could consist of applications that receive data from the network and display them, and of a management application that coordinates the display. The management application provides information to applications displaying data regarding the rate in which messages should be displayed and their display timing. The information results from monitoring of the arrival and the display processes of video and audio messages through traps defined on the contents of the MIBs. The following characteristics are monitored: (1) the number of messages that arrived and where not processed yet, by defining a trap that signals whenever the accumulated number of messages received minus the accumulated number of messages processed is over a certain threshold; (2) the relative pace in which the display of each participant is occurring, by defining a trap that signals whenever an application is displaying information that was sent and arrived a long time ago, and (3) the quality of real-time execution of the applications, by defining traps that signal whenever an application has missed more deadlines than expected. The first aspect being monitored avoids buffer overflow or underflow. The second detects situations in which the display of one of the conferences is outdated compared to others. The third detects applications whose performance are being severely affected by transient periods of overload. The management application monitors trap messages and synchronizes the applications displaying data. Example 2.12 shows part of the management application's trap conditions and exception handler ports specification in QuAL. QuAL introduces the **`monitor_cnd`** operator that consists of a monitoring condition and an exception handler port. A monitoring condition represents an application property that needs to be monitored (e.g., number of buffered messages on a specific communication end-point). The monitoring condition becomes true whenever the property being monitored reaches an unacceptable value (e.g., the number of buffered messages exceeded a certain threshold and a buffer overflow is likely to occur). In Example 2.12, the monitoring condition is based on values stored in the inport MIB. The inport MIB is identified by **`qInpAssocEntry`**, an object identifier according to the SMI standard. In this example, the MIBs and their respective fields are identified according to the definition in [florissi94]. The condition is monitoring the inport `video_inport` of the instantiation `partc1` of the QuAL process `video`. A message must be sent to the exception handler port `video_partc1_hndlr_port` whenever the number of buffered messages exceeds `LimitBufferedMss`. An exception handler port is specified after the keyword **`mgt_handler`**. The number of buffered messages is measured by subtracting the number of messages that were processed (stored in **`qinpProcessedMessages`**) and the number of messages that were discarded because they violated application-level QoS constraints (**`qinpAccumulatedApplQoSExceptions`** and **`qinpAccumulatedGroupApplQoSExceptions`**), from the number of messages that arrived in the order they were sent (**`qinpReceivedMessagesInSequence`**) plus the number of messages that did not arrive in the order they were sent (**`qinpReceivedMessagesOutSequence`**). A call to the **`monitor_cnd`** operator notifies the local QuAL's runtime SNMP agent that a message must be sent to the designated exception handler port whenever the condition evaluates to true. In Example 2.12, the management application first defines the monitoring conditions and associate exception handler ports, and then proceeds to listen to the designated ports, waiting for messages. Upon message arrival, the management application decides on the control instructions that need to be sent to the display applications. In the example, the management application sends a message to the display application asking the display application to increase the message display rate to avoid buffer overflow.

Example 2.12: Specifying Traps for Controlling A Multimedia Conference

```
% Management Application
monitor_cnd( qInpAssocEntry.qinpOSProcessIndex == partc1 &&
             qInpAssocEntry.qinpApplIndex == video &&
             qInpAssocEntry.qinpAssocIndex == video_inport &&
             (qInpAssocEntry.qinpReceivedMessagesInSequence +
               qInpAssocEntry.qinpReceivedMessagesOutSequence -
               qInpAssocEntry.qinpAccumulatedApplQoSExceptions -
               qInpAssocEntry.qinpAccumulatedGroupApplQoSExceptions -
              qInpAssocEntry.qinpProcessedMessages) > LimitBufferedMss)
mgt_handler video_partc1_hndlr_port;
```

```
while(TRUE) {
   switch(select(video_partc1_hndlr_port, video_partc2_hndlr_port)) {
   case 1:  % Send a message to partc1 display application to display
            % messages in a higher rate
   case 2:  % Send a message to partc2 display application to display
            % messages in a higher rate } ...
```

The QuAL generated MIBs expose the underlying management system to the configuration of the applications currently running, thus integrating application monitoring into general network management. It is neither efficient nor reasonable to manage all aspects of applications' distributed activities using only lower layer information because of the increasing gap between application-level abstractions and underlying-system entities providing services. The difficulty in managing applications this way increases dramatically as applications become more complex and the level of multiplexing between lower-level services and application-level abstractions become exponential.

Our management integration model provides a sound framework for distributing management responsibilities between general purpose management systems and application management systems. The MIBs generated comply with the SMI standard and QuAL's runtime includes a SNMP agent that answers SNMP requests. Thus, authorized external management stations can access these MIBs using the SNMP protocol and manage the underlying system accordingly. On the other hand, applications can be written in QuAL to manage QuAL applications. QuAL supports the SNMP get operations and the specification of trap conditions and associated exception handler ports that enable the development of application management stations in QuAL.

QuAL promotes application-customized management of QoS-demanding communications, critical for high-speed network (HSN) environments. HSN enable the development of widely diverse multimedia applications that differ in their communication requirements (e.g., audio applications require real-time delivery of data with no permutation whereas a collection of seismic data requires real-time delivery but allows the data to be permutated in the transmission). Each type of data being transmitted requires different control mechanisms to manage a communication. Thus, management becomes difficult, if not impossible, to generalize and needs to be done in an application-dependent manner.

Management is for the first time an integral part of the application development process as MIBs are structured and allocated as a result of the compilation process. Furthermore, they are automatically generated.

3 Technologies to Support QoS Assurance

In this section, we identify technologies necessary to support QoS assurance and some results we have achieved.

3.1 QoS-MIBs

We have designed an experimental version of the MIBs (also known as QoS-MIBs) that store management information acquired when monitoring QuAL applications and their QoS-demanding communications. The QoS-MIBs are defined in ASN.1 [rose91] and comply with the SMI [stallings93] standard. The complete definition can be found in [florissi94].

The design of the QoS-MIBs represents a first effort in defining the structure of the information necessary to manage communication QoS and real-time processing of general-purpose applications. As explained in Section 2.4, the information is organized in three MIBs: the application MIB, the inport MIB, and the outport MIB.

The application MIB contains general information regarding the status of applications (e.g., running, number of inbound connection addresses exported, etc.), their performance (e.g., number of times it succeed to execute in real-time), their non QoS-demanding connections (e.g., number of non QoS demanding active connections), and their QoS-demanding connections (e.g., number of active QoS-demanding connections).

The outport and inport MIBs contain detailed information regarding the QoS-demanding outbound and inbound connections, respectively. For outbound connections, QoS management information is captured by the following data: the QoS parameters negotiated with the network service provider at connection establishment time, the time in which the connection was established, the volume of data sent through this connection, the number of messages sent, the time in which the last message was sent, the number of times the connection went down, the time in which the connection had the last problem, the accumulated time spent in recovering from connection failures, and information regarding QuAL's network-level and application-level QoS monitoring and exception handling mechanisms. Information regarding QuAL's QoS monitoring and exception handling mechanism is captured by the number of network-level, single-stream application-level, and group application-level exception messages generated for the connection plus the time at which the last message was generated. The QoS of a communication can be evaluated using the information in the MIBs. For example, the average rate in which messages are sent can be calculated by dividing the number of messages sent by the elapsed time since connection establishment time until now. Similarly, the average time spent in recovering from connection failures can be calculated by dividing the accumulated time spent in recovering from connection failures by the number of times the connection went down.

Message loss and permutation during transmission complicate the capture of inbound communication QoS management information. We introduced the following convention to capture loss and permutation in QoS-MIBs. Messages in a connection are identified by an index that represents the order of the message in the sending flow. An outport QoS-MIB maintains, for each connection being monitored, a message counter that indicates the number of messages sent through the connection. The message counter of a connection is initially zero and is incremented every time a message is sent. Each message sent is stamped with the value of the message counter of the connection sending the message. The stamp represents the index of the message. Thus, the message counter of a connection captures the number of messages sent and the index of the last message sent. Due to loss and permutation, more information is needed on the inport QoS-MIBs to characterize the arrival of messages on an inbound connection. A distinction is made between messages that arrive *out of sequence* and messages that arrive *in sequence*, and separate statistics are maintained for the two type of messages flow. A message arrives out of sequence when it arrives late, that is, it arrives after the arrival of messages sent after it. A message arrives in sequence when it arrives before all messages that were sent after it. Loss or permutation is detected whenever a message arrives in sequence but before the arrival of messages sent before it. Consider, for instance, that the message arrival sequence of an inbound connection is the following, where messages are identified by their index: 1, 2, 5, 7, 4, 8, 3. The flow of messages that arrive in sequence consists of the messages 1, 2, 5, 7, and 8. The flow of messages that arrive out of sequence consists of the message 4 and 3. Loss or permutation are detected for the first time when message 5 arrives, and again when message 7 arrives. The messages 3, 4, and 6 are assumed to be lost. If they ever arrive, they are considered to arrive out of sequence. By distinguishing the two types of message flow and by keeping separate statistics for each one, inport QoS-MIBs capture statistics on loss and permutation.

Statistics on the loss and permutation rates of a connection can be calculated based on the information stored in the inport QoS-MIBs as follows. The index of the last message received in sequence represents a lower bound on the total number of messages that should have been received in sequence so far. An inport QoS-MIB maintains the number of messages received in sequence and the number of messages received out of sequence. The number of messages considered to be lost can be calculated by subtracting the number of messages that arrived in sequence plus the number of messages that arrived out of sequence from the index of the last message that arrived in sequence. The permutation rate can be calculated by dividing the number of messages that arrived out of sequence by the index of the last message that arrived in sequence. The permutation rate relative to the number of messages that arrived in sequence can be similarly calculated. All these statistics are based purely on the index of the last message that arrived in sequence. However, this index might not reflect the number of messages that should have arrived if the arrival rate is lower or higher than expected. More accurate calculus can be obtained by taking the status of the connection into account. For example, entries in the inport QoS-MIB also maintain the sending and arrival time of the last message that arrived in sequence, and information that can be used to calculate the average message sending rate and the connection's health. Statistics can be calculated based also on the expected number of messages that should have arrived and predict connection problems.

Another reason for distinguishing the flow of messages that arrive in sequence from the flow of messages that arrive out of sequence is that the semantics of the messages that arrive out of sequence depend on the application being monitored. Some applications (e.g., video transmission) discard messages that arrive out of sequence, whereas other applications (e.g., management applications that perform commutative operations on the contents of the data received) use messages that arrive out of sequence. By partitioning the information, statistics can be obtained on each type of message flow as well as be combined. This feature enables the capture of the semantics of messages that arrive out of sequence in an application dependent manner.

The main concern on designing the QoS-MIBs was to capture enough information on the communication flow of a connection to enable the analysis and study of its QoS. The automatic generation of these MIBs is an additional feature provided by QuAL's compiler and runtime. However, any application can be instrumented to generate and access these MIBs. We are in the process of instrumenting the XUNET multimedia conferencing application to generate these MIBs, in order to experiment with the QoS-MIBs design. We also intend to use the data gathered in the MIBs to implement an application management system to monitor the conferencing application.

3.2 Transport-Layer Design for QoS Assurance

We address the problem of how QuAL communication QoS specifications in terms of constraints are translated into transport-layer services that accomplish them. We make a first attempt in analyzing concrete services that we need from the layers below (e.g., network-layer and transport-layer) and that these layers are capable of delivering in order to achieve the required QoS. Traditional types of transport-layer services are *connection-oriented*, *connectionless*, *reliable* and *unreliable* services. To provide a connection-oriented service, a path is chosen between an origin and a target and buffering resources at each node in the path chosen are allocated. All the messages travel through the same path. Flow control protocols guarantee that messages are sent at the same rate they can be received. To provide a connectionless service, on the other hand, no path nor resources are allocated. For each message, an origin selects a node and sends the message to the chosen node. This choice is based on finding the least expensive path to the target, where expensiveness may depend on physical distance, system load, or any other metric. This process is repeated at each node, until the message reaches the desired target. Thus, inter-related messages may follow complete different paths. Reliable communications use error control protocols to recover lost messages, whereas unreliable do not.

Neither connection-oriented, connectionless, reliable, unreliable services nor any pure combination of them has been shown suitable for real-time delivery of messages. High-speed networks changed many of the invariants on which current network protocols are based, thus undermining their suitability. First, the lowering of the transmission speeds made visible the propagation delay at the links. Thus, old flow and error control protocols based on feedback operations are not suitable any more. By the time the feedback reaches the source, the information it brings might be outdated and might not be used to predict future behaviors. Second, processing speeds did not scale with transmission speeds, thus any processing inside the network ought to be minimized. Third, new applications that were unfeasible in conventional networks have become a reality (e.g., multimedia conferencing, virtual realities, etc.). Fourth, some network QoS parameters, such as bounds for end-to-end delay and transmission and loss rate, need to be abstracted at the application-layer. Upcoming applications have particular demands and quality assessments on the services the network provides, spawning new dimensions on how applications interact with network resources.

In the design of QuAL and its runtime, we assume that the following services are delivered by the transport-layer:

- The transport-layer translates communication rate and peak rate measures into the amount of bandwidth and buffering resources allocated and managed at each network entity involved in the communication. Nevertheless, no flow control is employed. A node sends messages to another node without knowing if the receiving node has received all messages previously sent or if it has processed the received ones to free buffering space for the new coming. This feature is provided by ST-II.
- Processing activities at each node in a connection are scheduled to guarantee that the sum of the queueing and processing delays at each node in the path does not exceed the transmission end-to-end delay negotiated at connection establishment time. ST-II makes a best effort in achieving this feature by prioritizing ST-II packets service over other IP packets service. However, there is no prioritization between the service of the ST-II connections themselves. Thus, communication delay must be monitored at the destination and applications must tune their message processing rate based on the current delay.
- The transport-layer makes a best effort (not necessarily guarantees) in assuring a communication probabilistic loss rate by, for example, sending several copies of the data being transmitted. ST-II supports this feature.

It is desirable that the transport-layer provides QoS re-negotiation on an open connection, but this feature can be overcome (as in the implementation of QuAL) by opening another connection with the new QoS requirements. Although part of the ST-II specification, there is no ST-II implementation that supports re-negotiation. The alternative solution of opening another connection is expensive and it may fail. Assume, for instance, that the rate of a connection should increase by 50% and that all the nodes in the connection have buffering and processing resources available for that. However, some nodes may not have enough resources to have both connections, the existing and the new one opened concurrently. Thus, the establishment of the new connection will not succeed unless the existing one is disconnected.

These issues lead to practical questions of what other services should be provided by the transport-protocols and how they should be designed to provide them or to improve the current services being offered. We have identified some areas in the design of transport-layers that need to be examined and that we expect to contribute in the thesis:

- *Scheduling of processing activities at network nodes*: Processing activities should be scheduled at every network entity (e.g., router) to prioritize processing on a per-connection basis, according to their QoS demands. If the scheduling is not performed properly, the processing overhead of certain communications (e.g., a file transfer) may incur unbound queueing delays for other communication streams. ATM networks [deprycker93] address this problem by making all the packets equally sized and by classifying packets as either priority packets or non priority packets. Priority packets have priority on the use of buffering space and processing resources over non priority packets. We intend to study *finer-grained adaptive* scheduling policies for processing activities in the network and to analyze the processing overhead they incur. Finer-grained policies classify packets in more than two classes, based on the particular QoS demands of each communication. Adaptive policies cause the dynamic change of the prioritization to recover from transient periods of degradation (e.g., if the queueing delay at one node is very high, the priority of the packet would be immediately increased to decrease its queueing delay at next nodes, compensating the extra delay).
- *Monitoring Services*: Since small degradations on the services provided by the transport-layer (e.g., a high delay for a single audio message) may incur large degradations on the performance of applications (e.g., display of video messages is delayed to control lip-sync, disrupting the video flow), applications must monitor the services provided by the transport-layer to immediate control the effect and recover from service degradations. The runtime of QuAL performs this monitoring and shelters from application developers the overhead needed to monitor services being provided by a transport-layer such as ST-II. We want to investigate the type of monitoring information that we need to get from the layers below and how the service access points can be extended to provide this information. Monitoring information provided by the transport layer decrease the overhead introduced at layers above to collect this information and promotes interoperability by standardizing the way the information should be collected.

3.2 Semantics

The challenges in providing a formal semantics for QuAL are to model the semantics of computations over real-time data streams (traditional approaches have mainly looked at how the execution is affected by the messages order of arrival). In this section we describe the approach that will be used in the thesis to formally define the semantics of the language constructs we are adding to handle QoS. The main goal in defining this formalism is to specify the semantics of the CMPs (as defined in Section 2.3) and to provide a framework for application developers to specify the semantics of their own Application CMPs (ACMPs). ACMPs implement the monitoring expressed through the application-level QoS constraints and their associated monitoring functions. Given the specification of the semantics, further work can be performed to analyze the properties of the sequences of messages communicated via such QoS control mechanism. The semantics for the real-time language constructs will not be described, since they have been the subject of previous works done in this area.

The unique feature of our approach is the use of Communicating Sequential Processes [hoare78] (CSP) for the characterization of communication temporal properties and for the specification of unreliable communication features (e.g., loss and permutation).

In CSP, inter-process communication is described as a special synchronous interaction, that is, a special event that requires simultaneous participation of both processes involved. Let the alphabet of a process be defined as the set of all possible events that a process can engage. If P and Q are processes with the same alphabet, the notation $P \| Q$ denotes the process which behaves like the system composed of processes P and Q interacting in lock-step synchronization. If the processes have different alphabets, events that are in both their alphabets require simultaneous participation of both P and Q. However, events in the alphabet of P but not in the alphabet of Q are not of concern to Q, which is physically incapable of controlling or even noticing them. Such events may occur independently of Q whenever P engages in them, and vice-versa. A communication is a special event described by a pair $c.v$, where c is the name of the channel on which the communication takes place and v is the value of the message which passes. Thus, a process that first outputs v on channel c and then behaves like P is defined as: $(c!v \rightarrow P) = (c.v \rightarrow P)$. Similarly, a process which is initially prepared to input any value x communicable on the channel c, and then behaves like $P(x)$, is defined as: $(c?x \rightarrow P(x))$. Let P and Q be processes and let c be a channel used for output by P and for input by Q. When these processes are composed concurrently in the system $P \| Q$, a communication can occur only when both processes engage simultaneously in that event, i.e., whenever P outputs a value v on channel c, and Q simultaneously inputs the same value.

We first characterize temporal behaviors using CSP. We introduce a clock process which ticks every interval of x ms, where x defines the duration of atomic intervals. Ticks are events of the clock process defined as follows: $C : tick_1 \rightarrow tick_2 \ldots tick_n \ldots$ We will only consider the systems that include one and only one clock process and that all other processes composing the system interact in lock-step synchronization with the clock. Events engaged by any process in the system, except for the clock process, that take at most x ms to execute are defined as atomic events. All other events are composed events represented by a sequence of atomic events. In this framework, a QuAL application or system Q' is represented by: $Q' \| C = P_1 \| P_2 \| \ldots \| P_n \| C$, where each process P_i is of the form $tick_j \rightarrow e^i_j \rightarrow$

$tick_{j+1} \rightarrow e^i{}_{j+1} \rightarrow \ldots$, and $e^i{}_j$ represents an atomic event of the process P_i that is executed between the interval from $tick_j$ to $tick_{j+1}$. In a $Q' \parallel C$ system, inter-process communication can only occur if one process is prepared to receive data in the same time interval the other is receiving the data.

We then proceed to represent a reliable asynchronous communication. We want to represent the propagation delay introduced by data communication over a reliable network that delivers messages in the order they enter the network. A system consisting of two QuAL processes *P* and *Q* communicating over a reliable connection is represented by the composed system: $P \parallel Q \parallel C \parallel N$, where *N* represents the reliable connection between processes *P* and *Q*. Let $P_{<s>}$ represent a process *P* that stores a data sequence *s*, *in* be the output channel used by *P* and the input channel used by *N*, and *out* be the output channel used by *N* and the input channel used by *Q*. Some additional operators need to be introduced before describing *N*. The symbol $\prod$ represents non-determinism, that is, $P \prod Q$ represents the system that either behaves as *P* or as *Q* in a non-deterministic way. The operator | represents a choice, that is, $Q = (a \mid b) \rightarrow P$ represents the process *Q* that either engages on *a* or on *b* and then behaves as *P*. When *Q* is put together with another process that can also engage in the event *a* but not in the event *b*, this means that *Q* will engage on *a* if the other process can also engage on *a*, or *Q* will engage on *b*, otherwise. *nop* represents the null operation. The only operator that does not belong originally to CSP and that will be used in defining *N* is the operator $\Rightarrow$. This operator represents the fact that a process can execute in parallel the input of a symbol and the output of another through two different channels, as an atomic event. The process *N* can then be represented as follows:

$$N_{<>} = tick_j \rightarrow in?x \rightarrow N_{<x>}$$

$$N_{<x>\wedge s} = tick_j \rightarrow (((in?y \mid nop_{y=<>}) \Rightarrow ((out!x \rightarrow N_{<s>\wedge y}) \mid (nop \rightarrow N_{x\wedge<s>\wedge y}))) \prod$$
$$((in?y \mid nop_{y=<>}) \Rightarrow (N_{x\wedge<s>\wedge y})))$$

Thus, *N* is a process with infinite storage capacity that is always, at an arbitrary $tick_j$, capable of engaging in an input event. If the process sending data is not willing to engage in such operation, *N* performs no operation (*nop*) in this channel. In parallel, it is non-deterministic whether *N* will be willing to engage in an output operation or not. However, when *N* is capable of engaging in an output operation, it only does so if the processes connected to the output channel is also willing to engage in a receive operation. Otherwise, it performs a *nop* operation in the output channel. The non-determinism characterizes the arbitrary delay introduced with traffic over a network. Note that the network takes at least an atomic period of time (*x* ms) to transmit. That is, the data that is input after a $tick_j$ is only output after the $tick_{j+1}$ has occurred. This design has also modeled in a very simplistic way the asynchrony of the communication. The sender process *P* performs a synchronous communication operation with process *N*, but not with process *Q*, the ultimate destination of the message. This approach models an arbitrary delay, but more accurate delays can also be captured. Process N can be designed to engage in an output event no longer than a certain interval of time after the arrival time of the next data that must be output.

We extend the basic framework to include the semantics of a QoS-dependable asynchronous communication. Initially, the communication process will include arbitrary loss and permutation factors. This can be achieved by intercepting the communication between process *N* and process *Q* with a third process *PL*. *PL* is always capable of receiving data from *N* and, whenever it has data stored, is also capable of sending data to Q. However, not all data received is stored (representing loss) and the data stored might not be sent to Q in the order in which it was received from N (representing permutation). This is in contrast with the way N delivered data to Q in the system previously described. N delivered data to Q in the sequence (order) in which it received from P. In designing PL, the choices of which data to store and send are represented by probabilistic distributions that better capture the actual QoS parameters allocated for a particular connection.

Given the specification of the communication semantics, further work can be performed to analyze and verify the properties of the sequence of messages communicated using QuAL's QoS-dependable communication mechanism and the semantics of the applications that use them. Through the semantics of the communication, the temporal properties of the communicated messages (e.g., sending, arrival and processing times) can be estimated, as well as their probability of loss or permutation. The semantics of the communicating applications can then be verified taking into consideration their temporal dependencies on the data being transmitted. The semantics analysis may detect situations in which applications could deadlock waiting for data that would never arrive on time, for example. It may also be used to study the temporal properties of applications to analyze the quality of the processing timing of the data being transmitted.

3.4 Filters

Monitoring and associated exception handling mechanisms written in QuAL define filters that actuate on streams being communicated among processes, similar to the filters usually studied in Control Theory [oppenhe83][kirk70]. We investigate how results from Control Theory can be applied to analyze properties (e.g., stability and convergence) of applications written in QuAL.

The QoS monitoring performed in QuAL is expressed as a function of time. The network-level QoS monitoring employed by QuAL's runtime checks whether the sending and arrival time of the messages being communicated comply with the negotiated QoS constraints defined at connection establishment time. Similarly, the monitoring functions defined by application developers to implement the application-level QoS constraints do not have access to the contents of the messages being transmitted, but only to their temporal properties. Furthermore, only the temporal behavior of the communications is recorded in QoS-MIBs. Upon violation of any QoS constraint, exception messages are generated and sent to designated exception handler ports.

The processes handling these exception messages or managing the QoS-MIBs control the application's performance. These processes analyze exception messages and MIB information, predict future QoS violations, and tune the application's flow of control to prevent potential problems. Consider for instance, the playout quality control problem. A process P displays messages and another process Q controls the display, providing P with feedback on when P should display the next message. Q analyzes the number of buffered messages (messages received but not processed) and the rate at which messages are being received and processed. Both parameters analyzed by Q are function of the temporal behavior of the messages. If there is a high oscillation on these values, Q studies the cause of the oscillation. Based again on the temporal behavior of the messages, Q determines if the best approach should be to request the sending process to change its sending rate, or if the network is not providing the services needed and the communication QoS parameters should be re-negotiated.

The QoS monitoring and control performed in QuAL characterizes the behavior of the traffic on ports, similarly to the effect of filters in Control Theory. In fact, the control performed by Q in the example above could be completely analyzed mathematically. The use of Control Theory enables the derivation of properties of the system such as *stability* and *convergence*. Stability measures how sensitive the management mechanism is to oscillations in the traffic. Convergence measures whether the system ever reaches a stable state or keeps trying to tune itself to changes of the temporal properties of the data.

We intend to study the following two dimensions of the problem:

- Identify a set of monitoring functions that have well-defined properties and provide them as a library to be used by application developers. These monitoring functions can be used to verify application-level QoS constraints or to implement the processes handling QoS exception messages. The mathematical properties of these functions will be studied in order to characterize the behavior of systems built using them. Application developers can use this set of functions to develop applications with well understood properties.
- Investigate the use of the mathematical framework in Control Theory as a software engineering that allows application developers to derive properties from programs written in QuAL. These tools can automatically study the mathematical properties of the monitoring and control functions used by the applications and derive the properties of the system.

3.5 Runtime Architecture

The runtime architecture of QuAL contains several modules that are explained in the appendixes, but in this section we summarize the main features of the two most challenging modules in QuAL: the real-time scheduler and the communication-support system.

In the description of QuAL's real-time scheduler, we describe how the runtime performs scheduleability analysis and supports real-time scheduling on top of a general purpose OS. The QuAL runtime can be efficiently implemented on top of a real-time OS (e.g., Split-Level Scheduler [govindan91]). However, these OS are usually embedded systems and not widely distributed. On the other hand, upcoming releases of general purpose OS (e.g., Solaris) introduce real-time features defined by the POSIX standard [posix90]. The compliance to these standards guarantee a maximum bound for the execution time of system calls and the locking of shared resources. These features give general purpose OS predictability and very high responsiveness. For our first prototype, we are studying the implementation of a real-time scheduler on top of UNIX version 4.3.

In the description of QuAL's communication-support system, we show how the runtime interacts with the ST-II transport protocol. Initially, the abstract QoS attributes are translated into ST-II service access point parameters. The runtime then interacts with the transport-layer entities in establishing the connection. All the details of the translation and negotiation are sheltered from application developers. Because ST-II provides no monitoring information (e.g., the propagation delay of packets or the average loss rate of the communication), the runtime adds control information to application's data being exchanged and implements all the monitoring needed. Application developers are only exposed to standard exception handling messages that are sent to designated ports.

4 Applications Implemented in QuAL

In this section, we discuss the implementation of two applications in QuAL. We first describe each application and identify their requirements. We then show how they can be implemented in QuAL. The final code is concise, efficient, and shelters from application-developers lower-level implementation detail.

4.1 Geology Application

Consider the acquisition and analysis of *time series* data for a seismic collection system. A time series is a collection of measurements that are sampled over small uniform intervals of time. Examples of time series are motion, vibration, and temperature measurements taken on crust plates over time to detect earthquakes and volcanoes, or to monitor the fluid dynamics of petroleum and ground water. The collection of data is inherently distributed. Sensors forward the data to relatively close general purpose machines that save the data into local disks. These disks are later physically transferred to more powerful servers where the data is downloaded and can be accessed and analyzed.

Geologists look forward to performing an interactive analysis of time series data as they are being captured. This involves transferring each time series in real-time from the machines placed close to the sensors to remote more powerful machines capable of filtering and analyzing each time series, as well as of analyzing the correlation among the dependent ones. These analyses usually involve complex arithmetic computations that are only effective on specialized processors. Such an application is only feasible over high-speed broadband networks due to the amount of data generated and the real-time nature of the analysis that needs to be performed.

This application illustrates some of the challenges in performing distributed computing over High Performance Computing and Communication (HPCC) environments. Initially, enough bandwidth needs to be allocated to guarantee that the data can be transmitted and in real-time. Although not desirable, the communication may tolerate a certain degree of loss and permutation, since the data is highly correlated. A lost sample can be substituted by a copy of the last data received and most of the mathematical calculus performed in their time-series are performed over a window of n samples, regardless of their particular order, tolerating some data permutation. Recoverability, though, is critical and needs to take place immediately. Machines located close to the sensors are mainly dedicated to local data storage and the eventual forwarding of data to remote machines, but not with data processing. The sampling rate of the local machines may be ideal for local data storage, since all the data must be recorded. However, it may be too fine grained for real-time analysis. Thus, it is desirable to control the rate of data sent to allocate network resources in a more effective manner. If no abnormality is detected, the remote analysis can be performed over only a fraction of the data sampled, as long as this fraction of data properly represents the time series. For instance, a good approach to reduce the sampling rate by half is to receive every other sample. However, upon detection of an eventual problem, resources must be re-negotiated and the remote analyses may require more information until the event is adequately identified.

In QuAL, this application can be easily implemented through the use of network- and application-level QoS specification and monitoring. First, consider the application that will run on the machines placed local to the sensors, defined in Example 4.1. The output port `sample_out` sends samples of type `sample_t` and has the following QoS requirements over its communication. It can tolerate any degree of loss or permutation, leaving it up to the receiver to further constrain this attribute; the rate of samples sent will never be higher than `SAMPLES`/sec, a value dictated by the sensor the application is communicating with. Other attributes such as recoverability time, maximum delay and inter-message delay are all undefined and are left up to the receiver to decide upon them. Any message reporting a network-level QoS violation is discarded, since no `net-hndlr` port is provided. The flow of messages on this port can be further monitored by any of the application-level QoS contracts specified: `rate_SR_half`, that sends every other message reducing the rate from `SAMPLES/sec` to half, `rate_SR_third`, that reduces the rate to one third, `rate_SR_fourth`, that reduces the rate to one fourth, or `NULL` that causes no reduction at all. No `app-hndlr` port is defined, so messages that are not sent are discarded. The application includes the definition of three functions that will actually implement the contracts supported on the ports. It is interesting to note that the sender side of this application is not concerned with what is actually happening at the communication level. Most of the QoS parameters are left for the receiver side to decide upon them and exception messages are simply discarded. The runtime system is responsible generate Communication Monitoring Processes (CMP) to manage the connection appropriately. Network-level constraints are managed by

Network CMP (NCMP) and application-level constraints are managed by Application-level CMP (ACMP).

The main body of the application is straightforward. Initially, the application associates functions to contract identifications, designating functions to implement the contracts. Then, bindings are exchanged as in Concert/C and the ports are bound. The local database is opened and the application proceeds to periodically getting samples from the sensor in real-time at the rate dictated by the sensor. The main flow of the application does not need to be aware of what ultimate network-level or application-level QoS attributes were actually chosen, since the run-time will be responsible for creating the respective CMPs to control and assure them. The local application acts as a mere server providing the data, but the data flow is finely tuned to comply with the receiver's demands. The receiver will be the one actually processing and interpreting the data, and therefore more concerned in monitoring and re-negotiating the QoS parameters dynamically. If the receiver requests a re-negotiation, the NCMP will handle the request. If the contract of the application-level QoS is also changed, the QuAL runtime will be responsible for terminating the current ACMPs and to create new ones.

Example 4.1: Geology Application: The Sender

```
% The Sender
#define SAMPLES 100
realtm {loss NULL; permt NULL;
        rate - sec SAMPLES; peak - sec SAMPLES;
        compl {rate_SR_half; rate_SR_third; rate_SR_fourth; NULL};}
handlers {net_handler manage_conn;}
receiveport {sample_t} sample_out;
% Function Definitions
int fnct_rate_SR_half(int i, time_tp t) {
   % If i is even return true, otherwise return false.
}
int fnct_rate_SR_third(int i, time_tp t) {
   % If i is a multiple of 3, return true, otherwise return false.
}
int fnct_rate_SR_fourth(int i, time_tp t) {
   % If i is a multiple of 4, return true, otherwise return false.
}
...
% Main Body
   % Designating Functions to Implement the Contracts
```

```
assg(sample_out, rate_SR_half, fnct_rate_SR_half);
assg(sample_out, rate_SR_third, fnct_rate_SR_third);
assg(sample_out, rate_SR_fourth, fnct_rate_SR_fourth);
% Binding Ports and Opening Local Database
...
% Sampling , Storing, and Sending Data
within(hard; periodic sec 1.0/SAMPLES;)
do {timeout (1.0/SAMPLES)
      {  % Read from the sensor
         % Save data on disk
         % Send data to receiver
         send(sample_out, data);
      }
      expired { /* Beep Alarms: Serious Deadline Missed! */ }
}
until(TRUE);
```

Example 4.2 describes the application that receives the samples and analyzes them. Loss and permutation are now given strict upper bounds. The receiver is not capable of processing 100 samples per second, so it gives lower bounds to the rate needed, mainly a quarter of the rate the sender is capable of sending, and requires that the sender complies with the contract `rate_SP_quarter` to uniformly discard messages. The input port complies with the contract `cnt_lp` that controls loss and permutation. Mainly, this contract guarantees that the application is only looking into a certain window of the stream of messages at a time, say all messages in the last 10 seconds. Since the connection is prone to loss and permutation, a message that was considered to be lost may simply arrive out of order much later than expected. This contract guarantees that such messages are discarded and that any message accessed through that port by the application was sent in the last 10 seconds. Strict bounds are given for the maximum delay for delivering a message, as well as for inter-message delay. The data is considered critical and recoverability should take no longer than 5 seconds. Unlike the sender application, the receiver application is concerned with the connection and specifies the input port `manage_conn` to receive any exception message that comes from the run-time reporting a QoS violation at the network-level.

The main body is simple. Initially a function is designated to implement the `cnt_lp` contract and the input port is exported to be bound. The application enters the real-time mode, waiting for a message either on the exception handler port or on the data port. Sample data is further processed by the function `process`, that returns a positive value if no abnormality is observed, or a negative value if a problem is observed and some action is needed. Probably the application may need to leave this main loop, re-negotiate

QoS parameters asking for more bandwidth, and perform a more detailed analysis. Exception messages are processed by the function `handle_exc` that filters and handles exceptions. The function checks the number of times the bounds for loss rate, permutation rate and maximum delay were violated. Whenever these events occur more often than an specified threshold, the application initiates a re-negotiation process, with more strict bounds. This main loop is repeated until either the `process` or `handle_exc` functions return a negative value.

Example 4.2: Geology Application: The Receiver

```
% The Receiver
realtm {loss 6; permt - sec SAMPLES/10;
        rate - sec 25; peak - sec 25; recovery sec 5;
        delay sec 1/SAMPLES; inter-delay sec 1;
        bnd_cmpl {rate_SP_quarter;};
        cmpl{cnt_lp;};}
handlers {net_handler manage_conn;};
receiveport {sample_t} *sample_in;
int fnct_cnt_lp (int i, time_tp send_time, time_tp arrv_time) {
   % If send_time was before than 10 seconds ago return false,
   %  otherwise return true.
}
int process(sample_t sample) { ... }
int handle_exc(exc_t message) { ... }
...
% Main Body
assg(sample_in, cnt_lp, fnct_cnt_lp);
...
within(soft; sporadic sec 1.0/SAMPLES;
        after (choice = select(sample_in, manage_conn));
        atEvent(choice = select(sample_in, manage_conn));)
do { switch(choice) {
          case 1:  timeout(sec AVERAGE_TIME_FOR_RECEIVE)
                   {  % Receive data
                      receive(sample_in, data);
                   }
                   expired { /* receive is taking longer than expected. */ }
                   timeout(sec  TIME_OUT_FOR_PROCESSING)
```

```
            {  % Process data
               normal = process(data);
            }
            expired  {  /* processing would take longer than expected. */
                        /* return an approximate result value */ }
            break;
   case 2:  timeout(sec  AVERAGE_TIME_FOR_RECEIVE)
            {  % Receive the Exception Message
               receive(manage_conn, exc);
            }
            expired  {  /* receive is taking  longer than expected */ }
            timeout(sec  TIME_OUT_FOR_PROCESSING_EXCEPTION)
            {  % Process the Exception Message
               normal = handle_exc(exc);
            }
            expired  {  /* processing would take longer than expected.
                           send an alarm! */ }
   }  }
until(normal);
```

4.2 Multimedia Communication

This application illustrates the use of inter-port application-level QoS monitoring. Consider the sending and receiving of audio and video streams that need to be synchronized. Synchronization needs to be monitored at the sending side to better utilize the network resources, as shown in Example 4.3. Since human beings are more sensitive to audio than to video, the audio stream has priority over the video stream. This means that the video stream may wait for the audio stream to synchronize, but the audio stream never waits for the video stream. In this particular application, the optimal rates for video and audio are of 30 frames per second and 160 audio messages per second, respectively. Each audio message consists of 50 audio samples of one byte each. This means that on average, for every group of 5,3 audio frames one video frame should be sent and this is the criteria used by a Group ACMP (GACMP) to monitor the traffic at the sending side. The process lets all audio messages to be ultimately sent and records the event. Video messages that are late, that is, that the application sends after its corresponding group of audio messages has already been sent and the sending of the next group has already started, are not sent but forwarded to the designated exception handler port. Video messages sent prematurely or on time are ultimately sent and it is up to the receiver process to buffer them. The sender consists of two light weight processes, one for sampling the audio device, and another to sample the video device. The process sampling the audio is a hard periodic process, whereas the one sampling the video is soft periodic, and therefore vulnerable to miss deadlines. This design choice promotes a better utilization of processing resources, since for soft tasks the scheduler works in a best effort basis to properly schedule the process without locking the resources preventing other processes that need to execute in a hard real-time mode to execute. The streams will go out of synchronization whenever the scheduler causes the soft process to miss a deadline. Another important aspect is the difference in the network-level QoS attribute values for the ports transferring audio and video. This helps to better explore the network resources, specially in cases where there could not be enough resources to provide the same quality to both streams requiring that priority be given to certain communications and quality be distributed hierarchically.

Example 4.3: Multimedia Communication: The Sender

```
% The Sender
enum {audio, video} syn_grp;
#define AUDIO 160
realtm  {noloss;  nopermt;
         rate - sec AUDIO; peak - sec AUDIO; recovery sec 5;
         delay sec 1/AUDIO; inter-delay sec 0.5;
         grp_cmpl{syn_grp[audio];};}
handlers {net_handler manage_audio_conn;}
receiveport {audio_t} *audio_out;

#define VIDEO 30
realtm {loss - sec 8; nopermt;
         rate - sec VIDEO; peak - sec VIDEO; recovery sec 5;
         delay sec 1/VIDEO; inter-delay sec 1;
         grp_cmpl{syn_grp[video];};}
handlers {app_handler video_not_sent; net_handler manage_video_conn;}
receiveport {video_t} *video_out;

int fnct_sync (syn_grp port, int i, time_tp send_time) {
   % If an audio message (port == audio),
   % Then increment counter of audio messages sent and return true;
   % Else , If (i * 5,3 ) is close enough to the number of
               audio messages already sent,
   %        Then return true, otherwise return false
```

```
}
...
% Main Body of Process Sampling Audio
...
assg_grp(syn_grp, fnct_sync);
within(hard; periodic sec 1.0/AUDIO;)
   do { /* Sample Audio Device */ }
   until(FALSE);
...
% Main Body of Process Sampling Video
...
within(soft; periodic sec 1.0/VIDEO;)
   do { /* Sample Video Device */ }
   until(FALSE);
```

Synchronization is also monitored at the receiving side to control the delay and the de-synchronization introduced by the network, as shown in Example 4.4. The audio connection does not tolerate loss or permutation, so no re-ordering needs to take place. The respective NCMP will monitor the connection and send a message to the `manage_audio_conn` port if the network is not delivering the QoS allocated. The video connection, on the other hand, allows loss and permutation to occur. The respective NCMP will send a message to the exception handler port `manage_video_conn`, if the communication QoS is violated. For instance, if the bound for the loss or permutation rate is violated. It is the responsibility of the application, however, to recover from message loss and permutation introduced by the communication that do not violate the bounds specified. Both audio and video ports are monitored by a GACMP. This GACMP studies the degree of de synchronization between the streams, producing statistical data that is used to better monitor the application and the underlying system. This process has immediate access to the send and arrival time of each audio and video message and has knowledge over the degree of their interdependencies, such as the rate of synchronization between them. This process can propagate this information through message passing to other application processes that can monitor the overall application performance. Besides performing statistical studies, the GACMP prioritize the service of audio stream over the video stream. All audio messages are immediately delivered to the application, whereas all video messages are forwarded to the exception handler port `buf_video`. This `buf_video` port belongs to another process created by the main application just to handle the video messages. This process discards late messages, buffers early ones, re-orders the ones out of order, replace lost ones with previous or subsequent messages in the sequence, and sends the ordered video messages to the main application. This auxiliary process, however, functions in a soft real-time mode, since the video frames are not as critical as the audio frames. This means that the quality of the image will depend on the system load, but the audio quality no. If the auxiliary process does not have the processing resources needed, than a deadline exception will be raised the next time it executes, causing the function `release_buffer_space` to be called. This function will remove from its buffer all buffered video messages that could not be sent to the main application on time and will reset its state.

Example 4.4: Multimedia Communication: The Receiver

```
% The Receiver
enum {audio, video} syn_rec_grp;
#define AUDIO 160
realtm {grp_bnd_cmpl {syn_grp[audio];};
            grp_cmpl {syn_rec_grp[audio];};}
handlers {net_handler manage_audio_conn;}
receiveport {audio_t} audio_in;

#define VIDEO 30
realtm {grp_bnd_cmpl {syn_grp[video];};
            grp_cmpl{syn_rec_grp[video];};}
handlers {net_handler manage_video_conn;
            grp_app_handler buf_video;}
receiveport {video_t} video_in;

int fnct_sync_rec_grp (syn_grp port, int i,
                       time_tp send_time, time_tp arr_time)
{
   % If an audio message (port == audio),
   % Then i update statistics and return true;
   % Else , update statistics and return false.
}
. . .
% Main Process
. . .
assg_grp(syn_rec_grp, fnct_sync_rec_grp);
within(hard; sporadic sec 1.0/(AUDIO+VIDEO);
         atEvent (choice = select(audio_in, aux_video));)
   do { switch(choice) {
```

```
            case 1: % Display audio
                  break;
            case 2: % Display Video
                  break;
          }
    until(FALSE);
...
% Main Body of Auxiliary Process
...
within(soft; sporadic sec 1.0/VIDEO;
         event = atEvent(select(buf_video) || atPeriod(1.0/VIDEO));)
   do { if(event == 1) {
         % Buffer or Discard video frame
         }
         else {
         % If the video frame that should be sent now has arrived, send it,
             Otherwise, send the previous frame or the next one.
         }
       }
   deadline { /* release_buffer_space */ }
   until(FALSE);
```

The main body is simple. Initially the application designates functions to implement the contracts, creates auxiliary processes, and binds the ports. Then, it engages in the main loop that consists of checking either for audio messages from the port connected to the remote sender or video messages connected to the auxiliary process. It is important to note that the main application does not deal with any synchronization aspect directly. All the synchronization and connection details are dealt with by the CMPs and the additional auxiliary process that buffers video messages. This same framework can be used to synchronize any two streams that present similar constraints, such as text and video, for instance, and can be easily extended to any number of streams. The framework is also general enough to be easily used to express any type of inter-dependencies between streams.

5 Related Work

In this section we review related work and position QuAL in this context. QuAL comprises work in three distinct areas: distributed computing, reviewed in Section 5.1, characterization and handling of QoS, reviewed in Section 5.2, and real-time programming, reviewed in Section 5.3.

5.1 Distributed Computing

A *distributed system* is an application that must be executed on a distributed computing environment (possibly heterogeneous). The *process model* [hoare78] is an abstraction that reflects main objects and their relationships in a distributed system. A computation is performed by autonomous processing units which interact and collaborate to solve a particular task by exchanging messages. In this section, we review current approaches for programming distributed systems in lieu of the process model and argue as to why Concert/C was chosen as our basic engine.

From an application programmer perspective, communication facilities are provided at levels as low as the transport layer [stallings94]. The most basic transport-layer service is the *socket* [stevens90] abstraction. Sockets are Service Access Points (SAPs) that entail transport-layer addresses used by processes to communicate. Sockets allow the specification of low level communication details, such as connectionless or connection-oriented service. No compile-time or run-time support is provided to verify more application-oriented features such as the type of information being exchanged, flow of control issues, etc. All these high-level features must be explicitly coded by the application developer. This requires a considerable level of expertise on transport-layer issues and results in non-reliable and complex codes. Furthermore, the topology of the processes in the application must be explicitly specified by the programmer in terms of the location and address of each process.

The Remote Procedure Call (RPC) abstraction [nelson81][soares92] was created to convey these issues. RPC extends the procedure call mechanism to a distributed environment, whereby a process can call a procedure that belongs to another process. Interprocess communication is achieved by using the syntax and semantics of a well accepted strongly typed language abstraction. RPC also constitutes a sound basis to move existing applications to the distributed system environment, thus supporting software reusability. RPC enables *location transparency*, whereby a process can communicate with another process using language-level identifiers, function references, without knowledge of transport-layer details. Also, the communication abstraction is simple: synchronous communication with the caller blocking until the result of the RPC is returned from the callee. The type of the data being exchanged are the procedure parameters and the compiler is able to type-check the communication.

A natural extension of this trend was to develop completely new languages that are especially suitable for distributed computing. Examples include many programming paradigms such as object-oriented

(Emerald [jul88]), logic for distributed computing (Prolog [schapiro86]), functional (Concurrent ML [reppy91]), and process-oriented (HERMES [strom91] and Ada [ada83]). A complete survey of programming languages paradigms for distributed computing can be found in [bal89], and an analysis of several paradigms for process interaction in distributed programs in [andrews91].

A compromise was necessary between the conflicting goals of having a new distributed programming language and being compatible with existing environments. Thus, existing programming environments and languages have been extended in several dimensions to support distributed computing at a higher-level. The most common approach is to provide RPC packages [RFC10057][kong90][dce91][soares92]. However, due to language details, the programmer is often exposed to a complex programming interface and has to perform several tasks in order to bring the program to a state in which RPC can be directly used. Usually, the process owning the remote procedure must register in the RPC runtime in the transport layer, and in the associated name servers. Only then can it listen for arriving calls. The process calling the procedure must invoke transport-layer services, name servers, and other RPC implementation services to locate the remote procedure and establish a remote binding. These features undermine location, syntax and semantics transparency.

Concert [yemini89][auerbach91] was developed to overcome these drawbacks. Concert is in reality a family of language extensions, being developed at IBM T. J. Watson Research Laboratory, to support distributed computing. The approach taken by Concert minimizes the learning overhead and does not incur the loss of location, syntax, and semantics transparency. Languages are extended with a concise set of new types and operators that support the process model while allowing reuse of existing code. The process model is supported directly within new languages by these extensions. Thus, it eliminates the multiple abstractions of language, OS, and add-on packages, and replaces them with a single, higher-level abstraction. In such an integrated system, the application developer uses a single programming interface, ignoring the details of how this interface is supported. It is the responsibility of the language designer and implementer to transparently map the Concert interface into low-level services provided by the language run-time system and underlying OS.

Concert also introduces the notion of interoperability whereby programs written in any of the Concert family of languages interoperate with each other and also with services written using conventional RPC packages. The interoperability is done by the language run-time system, and it is transparent to the application developer. The Concert run-time system shelters heterogeneity at machine-level, OS-level, and communication-protocol-level. As a consequence, a program written in one language can communicate with another program written in a different language, and running on a different OS through Concert. A Concert program is also interoperable with non-Concert programs running other inter-process communication protocols.

Concert/C is the language extension to C [kernighan88] to support the Concert approach for distributed computing. Concert/C has been prototyped and it supports a strongly-typed message-passing mechanism and RPC to specify inter-process communication integrated into the language.

Because of these features, we have chosen the Concert approach for distributed computing and Concert/C as the platform over which we build our extensions for the development of HPCC applications. A more detailed overview of Concert and Concert/C is presented in Section A of the appendix.

5.2 QoS

Inter-process communication abstractions are appropriate for traditional applications because they abstract communication delays that the programmer does not need to understand in order to perform its job. This is not the case for future HPCC applications because QoS parameters must be negotiated with the network and the application must be aware of periods in which the network is unable to provide them. Future communication abstractions must therefore explicitly enable specification, negotiation, and monitoring of such QoS parameters, while sheltering all communication details that are not relevant for the application. In this section, we review efforts that have been geared towards providing QoS to applications.

Provision of QoS at the transport layer has been the subject of much effort recently [forgie79]. Protocols at the transport layer are developed to provide end-to-end guaranteed service across a network. It is the responsibility of the application to exchange information among its peers to set up the QoS parameters, sometimes via another inter-process communication mechanism. The set up phase includes selecting the characteristics of the data flow between origin and target(s), identifying the SAP relevant to the specific transport protocol being used, and ensuring security, if necessary. Parameters that may be specified are bandwidth, delay, and reliability. Data is transmitted as part of the point-to-point or point-to-multipoint connections. During connection setup, paths to the destination are selected and the necessary network resources are reserved. Mechanisms at the transport layer require extra effort from the application developer to understand communication details, similar to what happens with the socket abstraction.

An example of a transport-layer protocol for QoS provision is the Stream Protocol Version 2 (ST-II) [forgie79]. The motivation for this protocol was to enhance the Internet Protocol [stevens90] for the establishment of QoS-dependable communication streams.

Transport-layer protocols that use ST-II have been developed to ease the establishment of communication streams in specific domains. For example, the Packet Video Protocol (PVP) [cole81] and the Network Voice Protocol (NVP) [cohen81] can be used directly by applications transmitting video and audio, respectively. PVP and NVP automatically choose the ST-II QoS parameters most appropriate for the transmission.

A more specialized transport protocol for audio and video digital communication across interconnected packet switched networks is described in [jeffay92]. This transport protocol restricts the services provided. The QoS parameters that can be specified are synchronization type between audio and video, number of frames played out of order, and end-to-end latency.

In addition to the complexity involved in the connection establishment and monitoring of such communications, applications that use transport-layer inter-process communication facilities are directly restricted to nodes that support the respective protocol. The Session Reservation Protocol (SRP) [anderson90] is a step towards overcoming this problem. SRP is a session-layer resource reservation protocol for guaranteed-performance communication in the Internet and it works independently of any particular transport protocol. SRP can be used with standard protocols, such as IP [postel81], or with new protocols, such as ST-II. A host implementing SRP can use IP when communicating with hosts not supporting SRP. SRP uses the DASH resource model [anderson90] to specify the reservation of *resource* (disk, CPU or network) capacity. SRP is directly responsible for reserving all network resources and can thus be viewed as a *network management protocol*.

Even session-layer protocols are low-level abstractions for application developers. The application-developer still needs to handle issues such as data representation across heterogeneous environments and inter-process synchronization model. In order to cope with these drawbacks, application-level support tools for specific domains were developed.

The Movie Control, Access, and Management (MCAM) [keller93] is an ISO [iso] application-layer architecture for handling video streams in a heterogeneous distributed environment. A *source* is any entity reading from an input device and producing a stream and a *sink* is an entity consuming a stream. Remote connections between sources and sinks in a heterogeneous environment are accomplished through a MCAM well-defined protocol. Information is passed between a movie service user and a movie service provider using service primitives.

The functional model of the system consists of four parts. The *directory system* is used to store and access distributed movie information. The *Equipment Control System* (ECS) allows the integrated handling of remote devices, e.g., to perform a camera zoom, or to adjust the volume of speakers. The *Stream Provider System* (SPS) provides to MCAM a plain stream service. The *MCAM system* interacts with other systems to provide services to an MCAM user. All services can be accessed at the SAP of the MCAM layer. Every play operation creates a *context* that carries information about the current value of the stream parameters supported. Such parameters are reliability (error-free or non-error-free), speed (retrieval and transmission), mode (fast or slow motion), quality (compression algorithm used), section (parts of the movie), and direction (forward or backward). All these parameters except the quality parameter can be modified by the user, if the movie is stored in a file. For live transmission, however, the user can only change the reliability, mode and quality. SPS is limited to video-specific applications and parameters negotiation is asymmetric. The service user specifies the stream parameters and the provider cannot restrict or negotiate them.

QuAL provides an application-level abstraction for the negotiation, establishment and management of QoS-dependable communications. The negotiation is completely symmetric. Both sender and receiver specify the QoS values desirable for the communication. The model guarantees that connections be opened only between senders and receivers that have matching QoS communication requirements. This is in contrast with MCAM, for instance, that the service user dictates the QoS parameters of the communication. Furthermore, QuAL provides an abstract model for the specification of QoS parameters that is independent of the communication protocol used, and also independent of the nature of the data being transmitted. This approach is general, it does not limit the domain of applications suitable to this framework, and it enables the bridging of heterogeneity at transport and session layers.

5.3 Real-time Language Constructs

One of the key issues in HPCC application development is how to specify the QoS that should be delivered by the underlying runtime system. Many such QoS requirements relate to temporal behaviors. It is thus necessary to use real-time language constructs to specify such QoS demands. In this section, we present a brief survey of real-time language constructs.

There has been a significant amount of research done in providing language-level support for specifying real-time constraints in task executions. In such languages, one specifies the constraints of the tasks to be executed and the underlying system is responsible for reserving the resources, and for scheduling the tasks accordingly.

Real-time constraints are classified as either *hard* or *soft*. Hard real-time constraints have to be met, whereas soft real-time constraints may be eventually violated (some recovery mechanism to handle violations must be provided). To guarantee that hard real-time constraints are met, a worst case performance analysis of the tasks to be executed must be performed at compile time. The analysis determines each task workload and is used for the allocation of resources. Based on workloads and constraints, the system must find an execution schedule (scheduleability analysis). The fact that the execution time of all program segments must be known at compile-time imposes severe restrictions on the language constructs that can be used as well as on the underlying OS. Examples of constructs that make such an estimate impossible are *while* loops. The underlying system must also provide an upper bound for OS calls, such as input and output operations. The scheduleability analysis problem is complex, and in some cases intractable [ullman73] [garey75].

Real-time systems differ on the real-time constraints that can be specified, on the type of scheduleability analysis they employ, and on recovery mechanisms they support. A detailed survey of real-time languages can be found in [halang91].

Since real-time system implementation imposes so many challenges, existing languages are customized to solve specific problems, without stretching all real-time language requirements. Some examples of soft real-time languages are Real-Time Language/2 (RTL/2) [barnes76] for industrial process control, Process and Experiment Automation Real-time Language (PEARL) [kap77] designed by a group of German researchers from research institutes and industrial firms, ILIAD [pickett79] designed by General Motors' Research Laboratories, and PORTAL [nageli79] designed to be used in system programming and real-time process control. All these languages implement heuristic scheduleability analysis (thus causing deadlines to be missed) and weak exception-handling mechanism for the handling and recovery of such violations.

Ada [ada83], on the other hand, is a general purpose programming language that includes real-time capabilities. Ada was designed as the result of a procurement exercise by the U.S. Department of Defense and it is expected to become the most used real-time language in the near future. Being a typical design-by-the-committee product, it includes just about every feature conceivable in a modern language. Ada hardly makes any provisions for scheduleability analysis and, as a result of its size and complexity, requires large amounts of run-time support. The only way for expressing time dependencies in Ada is to delay the execution of tasks by specified periods. Thus, Ada programmers have to hand-tune task timing dependencies. Furthermore, it does not prevent deadlocks during shared-memory access and resources are allocated in a first-in-first-out basis. The multitasking model in Ada does not deliver predictable execution.

None of the recently developed hard real-time languages is in wide use, and are classified as experimental. Examples include TOMAL [kieburtz76], DICON [lee84] [lee85], ORE [donner88], FLEX [lin88], Real-time Mentat [grimshaw89], RTC++ [ishikawa90], and Real-Time Euclid [kligerman86] [stoyenko87]. We choose to overview Real-time Euclid as a representative example of such languages, since it meets all the standard requirements to support hard real-time programming.

Real-time Euclid is a descendant of Concurrent Euclid [cordy81]. Programs can always be analyzed for guaranteed scheduleability, and use a structured exception handling mechanism. Inter-process synchronization mechanisms, such as monitors and signals, are analyzed to study the timing dependencies among processes and to prevent deadlocks. Statements to wait on signals are extended to specify time-outs and corresponding exception handlers. Default system-level exception handlers are defined for standard exceptions. Loops are restricted to iterate no more than a compile-time known number of iterations. There is no provision for dynamic data structures, since it may introduce unpredictability in both time and storage requirements. No recursion is permitted, since it not possible to calculate an upper-bound on the number of times the function will be called.

The scheduleability analysis is completely performed during compilation. Real-Time Euclid processes can be either *periodic*, or *aperiodic*. Periodic processes are those which have to execute at regular intervals, and must be completed before the next interval is due. Aperiodic processes are asynchronous processes that have only soft deadlines and for which no minimum inter-execution time interval is known. Aperiodic processes can be activated by system interrupts, by other processes or by timers. This information is used by the scheduleability analyzer to decide if the entire system of processes is scheduleable and by the run-time system to properly schedule processes.

An extension of PEARL to support hard-real time programming has been proposed in [halang91]. The main goal is to bring the good programming practices of Real-Time Euclid from the experimental to the industrial world. PEARL is still in the design phase, and it requires a specialized underlying OS to support its implementation. In order to guarantee the compliance of real-time constraints, resources are allocated to handle the worst case scenario, that is, when the system has the highest work load. This pessimistic approach, although necessary, may lead to low system use, especially when the system load fluctuates highly. To overcome this problem, PEARL uses the concept of *imprecise results* [lin87] to provide graceful system degradation. Imprecise results are defined as the most recent partial results. This idea only works for monotonicaly improving computations. By making results of poorer quality available when the results of desirable quality cannot be obtained in time, real-time services of potentially inferior quality are provided in a timely fashion.

QuAL builds a general-purpose high-level real-time language suitable for hard and soft real-time programming with predictable behavior. Similar to Real-time Euclid and extended PEARL, it provides high-level language constructs for the specification of process scheduling and control, and time constrained behavior. On the other hand, QuAL puts less restrictions on the underlying supporting system without sacrificing behavior predictability.

6 Final Remarks and Work Schedule

New applications enabled by HSNs demand substantial changes in the way HPCC distributed applications will be implemented. QuAL enables an application developer to abstract the network functionality in a way that is a trade-off between simplicity and maximum network functionality. HPCC applications have dramatically orthogonal requirements, which undermine any effort to completely abstract standard network operations at the application level, as is done in current distributed languages. Nevertheless, non-standard network operations tend to further complicate the application developer job.

In QuAL we have found a mid-field blending of complexity and functionality by extending the existing process model in the Concert/C language in four directions. First, network QoS parameters may be specified for the proper allocation of resources in the network. Second, the concept of a port is extended to include real-time behaviors. Third, real-time operators are defined that enable the proper scheduling of processes. Fourth, QuAL supports automatic generation of network management information.

These features render QuAL a first tool towards the development of appropriate software engineering environments for the development of HPCC applications.

We propose to conduct the work towards the thesis in the following way:

- Current Status:
    - Network-level QoS negotiation;
    - Scheduling of hard periodic tasks in a single address space;

- March 94:
    - Network-level QoS monitoring;
    - Scheduling of soft and hard tasks in a single address space;
    - Network-level QoS re-negotiation;
    - Application-level QoS negotiation;
- May 94:
    - Design of scheduling of tasks in multiple address spaces;
    - Testing and measurements of real-time;
- September 94:
    - Temporal-behavior of messages;
    - Single Stream application-level QoS monitoring;
    - Group application-level QoS negotiation;
    - Automatic generation of QoS-MIBs;
- December 94:
    - Writing;
    - Time-based access to ports;
    - Group app-level QoS monitoring;
- May 95:
    - Writing;
    - Testing and integration.

Appendix

QuAL is compatible with the Concert approach for performing distributed computing and it is an extension of the Concert/C language. In Section A, we review Concert and Concert/C. In Section B, we describe the syntax and informal semantics of QuAL language constructs. In Section C, we describe the architecture of the QuAL communication-support system and identify, in greater detail, the services QuAL's runtime system provides, the services the runtime expects to be delivered from the transport-layer, and how the runtime interacts with the transport-layer. In Section D, we describe the modules that form the entire run-time architecture of QuAL, and their functionalities. We give details on the design of the QoS monitoring mechanism and on the implementation of the real-time scheduler.

A Concert and Concert/C

Concert [yemini89][auerbach91] is a family of language extensions to support distributed computing using the *process model* [hoare78]. In the process model, *processes* are units of execution that communicate and synchronize with one another through message-passing.

In the approach used by Concert, the process model is supported directly within a new language by adding extensions to the language. These extensions integrate in a language the concepts of processes and ports, and a set of operations. A process is mapped into any active entity that performs computations. Ports are communication end-points that can be further classified as *input ports* (*inports*), or *output ports* (*outports*), depending on whether they are receiving or sending messages, respectively. An inport contains a queue to store messages that arrive and cannot be processed immediately (a process receives messages in their order of arrival). An outport contains a *binding*, that is, a capability of placing messages on an inport queue. The type of a port is defined by the type of the messages it sends or receives, and only ports of the same type can be bound.

The set of operations integrated in a language enables the creation and termination of processes, creation and termination of bindings, and the actual communication between processes. Any language in the Concert family will support these operators. Two forms of Inter-Process Communication (IPC) are supported: *asynchronous* and *synchronous* message passing. In the asynchronous mode, a process sends a message and continues executing. The receiving process can dequeue the message after it arrives. The synchronous mode is equivalent to a RPC [nelson81][soares92]. RPCs in the Concert model are made transparently through function pointers. The interface for performing procedure calls through function pointers is the same for both local and remote calls. In the remote case, however, the function pointer contains a binding that points to the remote function. The process making the remote call blocks until the process receiving the call executes the function and returns the results.

Concert/C [auerbach92] is a new language that extends C [kernighan88] to support distributed computing according to the approach defined by Concert. A Concert/C process is an executing C program. Concert/C introduces input ports as a new data type. Ports can be declared as being functions that can be called from another process (*functionports*), or simply as plain ports (*receiveports*). Functionports are defined by adding the keyword `port`[1] to a function declaration. Receiveports are declared as follows:

`receiveport { <message_type> } <identifier>.`

The `<message_type>` identifies the type of messages received through the port identified by the `<identifier>`.

[1]The symbol - is used to separate the lower from the upper limits of the interval being specified. When the lower or the upper bound are not specified a value of 0 or inifinite is assumed, respectively.

A binding is simply a pointer to an input port. A binding can reference any port of the type it points to. When a process is created, the parent process obtains an initialized binding (pointer) to an input port in the child process that can be used by the parent to initiate communications with the child. This input port of the child process is known as *initialport* . The keyword `initial` is used to identify the respective initialport in a child process.

Two operators are provided to check whether there are enqueued messages on a particular inport. Operator `select` accepts a list of ports as arguments and blocks until at least one of the ports has a message. The value returned by this operator is the index (that is, the position of the port in the argument list) of a busy port. Operator `poll` is a non-blocking version of `select`, which returns the value 0 if all the queues are empty. The `receive` operator is used to dequeue a message from an inport. If the queue is empty, `receive` blocks until a message is enqueued. A process that defines a functionport uses the operator `accept` to receive a message sent to this port, execute the function associated to it, and return the results to the sender. `accept` accepts a list of functionports and waits until at least one of them has a message. It then receives a message representing a function call (called a *callmessage*) from a non-empty functionport, invokes the associated function with parameters supplied from the message, and returns results from the function invocation to the calling process.

The operator `send` supports asynchronous message passing, returning as soon as the underlying system supporting Concert/C has copied the message to its internal buffer.

Concert/C supports process management, that is, creation and termination. Suppose that a program has been compiled by the Concert/C compiler and stored in the file `serv_sql4` on the machine `cs.columbia.edu`. Another process can create, and later terminate this process as follows:

```
[[ program server "serv_sql4"
   newspace host "cs.columbia.edu"; ]]
main()
{  ...
   server_handle = create(server, &init_port);
   ...
   terminate(server_handle);
}
```

The declaration inside the double brackets consists of a *distributed linking declaration*, that is, a declaration that controls the instantiation and linking of distributed programs. In the example, the declaration defines the variable `server` of type `prog_t`. The type `prog_t` stores a program description that can be instantiated into a running process using operator `create`. The variable will contain the object code from file `serv_sql4`. The `create` operator stores in the memory position designated by its second argument (the address of the variable `init_port`) the binding (pointer) to the initialport of the process created. The `create` operator returns a reference to the child process created (stored in the variable `server_handle`). The operator `terminate` terminates a process. It accepts as argument the process reference returned by the `create` operator.

B Syntax and Informal Semantics of QuAL Language Constructs

In this Section, we describe the syntax and informal semantics of the language constructs introduced by QuAL. In Section B.1, we describe the specification of QoS measures and processing constraints in QuAL. In Section B.2, we describe the operators introduced to access the temporal properties of messages (sending, arrival, and processing times) and to prioritize message processing based on this timing information. In Section B.3, we describe the specification of QoS measures and processing constraints control in QuAL. In Section B.4, we describe the operators added to support the access to the QoS-MIBs generated. When defining the syntax, the following convention is used. Keywords and constructs from QuAL are written in **bold** face, from Concert/C are underlined, and from C are plain text.

B.1 Specification of QoS Measures and Processing Constraints

In this section, we describe the syntax for the specification of QoS measures, detailed in Sections B.1.1 and B.1.2, and processing constraints, detailed in Section B.1.3, in QuAL.

In QuAL, QoS measures are part of the specification of the type of a port, so that constraints on the communication QoS on that interface can be assured, monitored, and controlled. To support the specification of QoS measures, QuAL extends the Concert/C port data type into the real-time port data type. The general form for the specification of real-time ports is as follows:

```
<real-time-port>  ::= realtm  [{<real-time-port-type-attributes>}]
    <handlers>  <concert-port-definition>;

<handlers> :: = [handlers  {<list-of-handlers>}]
<concert-port-definition> ::= <concert-outport-definition>  |
    <concert-inport-definition>
```

The keyword **`realtm`** classifies the port as a real-time port and causes the QuAL runtime to maintain the temporal properties (sending, arrival, and processing times) of the messages communicated through them. The QoS measures are specified through the **`<real-time-port-type-attributes>`**, explained in Sections B.1.1 and B.1.2, and define how the temporal properties of the messages are to be monitored. When these measures are not specified, the temporal properties of real-time port messages are maintained but not monitored by the runtime. In any case, the temporal properties of real-time port messages can always be accessed and used to prioritize message processing. In Section B.2, we describe the

QuAL operators that make this access and this prioritization possible. The **<handlers>** clause contains the specification of the exception handler ports associated with the port being specified and are explained in Section B.3. While the QoS attributes are part of the type of a real-time port, the exception handler ports declaration is only a specifier of the port being defined and are not part of the type. The **<concert-port-definition>** part specifies, as in Concert/C, the type of the messages exchanged through that port and if the port is an input port or an output port. Real-time ports are completely backwards compatible with Concert/C ports, supporting all access operations defined for them.

The general form for the specification of the attributes is as follows:

```
<real-time-port-type-attributes> ::= <net-qos>  <app-qos>
```

The network-level QoS measures are specified through the **<net-qos>** attributes, explained in Section B.1.1, and the application-level QoS measures are specified through the **<app-qos>** attributes, explained in Section B.1.2.

The processing timing constraints specification in QuAL is explained in Section B.1.3.

B.1.1 Specification of Network-level QoS Measures

The general form for the specification of **<net-qos>** attributes is as follows:

```
<net-qos> ::= [<loss>] [<permutation>] [<rate>] [<peak-rate>]
    [<total-delay>] [<jitter>] [<recovery-time>] [<excep-handler>]

<loss> ::= noloss | loss NULL | loss [<range>]
<permutation> ::= nopermt | permt NULL | permt [<range>]
<rate> ::= rate <range>
<peak-rate> ::= peak <range>
<total-delay> ::= delay <range>
<jitter> ::= inter-delay <range>
<recovery-time> ::= recovery <time-expression> | recovery NULL

<range> ::= <time-expression> - <time-expression> |
    <time-expression> - | - <time-expression>
<time-expression> ::= [ms | sec | min | hr] <constant-expression>
```

The specification of any of these attributes is optional. When an attribute is not specified, the port receives the respective QoS type of a communication in Concert/C. That is, no tolerance for loss or permutation and unbounded rates, delay and recovery time intervals. A **NULL** value means that the attribute can assume any value. The range of values for this attribute can be further *refined* if the port is bound to another port that has more restrict constraints. The refinement process was discussed Section 2.1.1. Resources are allocated in order to provide a connection that complies with these parameters.

B.1.2 Application-level QoS Specification and Monitoring

The general form for the specification of application-level QoS attributes is as follows:

```
<appl-qos> ::=
    <port-constraints>  <inter-port-constraints>
<port-constraints> ::= [bnd_cmpl {<contract-id-list>};]
    [cmpl {<contract-id-list>};]
<contract-id-list>  ::=  {<contract-id>}+
<contract-id> ::= <identifier>

<inter-port-constraints> ::= [grp_bnd_cmpl {<enum-id-list>};]
    [grp_cmpl {<enum-id-list>[<enum-element-identifier>]};]
<enum-id-list>  ::= {<enum-identifier>}+
```

Two features are specified through the **<port-constraints>** clause: (1) the port is capable of complying to the contracts identified in the **<contract-id-list>** following the keyword **cmpl**, and (2) the port can only be connected to ports capable of complying with one of the contracts specified in the **<contract-id-list>** following the keyword **bnd_cmpl.** While connected, a port complies with only one of the contracts specified in a **<contract-id-list>**. The specific contract is chosen by the other port that is connected to this one. Messages are monitored in a real-time fashion. Messages violating the contract chosen are either discarded or forward to a designated exception handler port. The specification of exception handler ports is described in Section B.3. Similarly, inter-port constraints are specified through the **<inter-port-constraints>** clause. However, the contract identification for inter-port constraints is an enumeration identifier. Each element in the enumeration represents one of the ports that are inter-related. When a port is defined as being capable of complying with an inter-port QoS, the declaration must identify which element of the enumeration that port will represent. This is specified by the **<enum-element-identifier>** inside the brackets.

A monitoring function is linked to a port contract identification through the QuAL operators **assg** and **assg_grp**, as follows:

```
assg(<port-symbol>, <contract-id>, <function-reference>);
assg_grp(<enum-identifier>, <function-reference>);
```

The operator **assg** is used for contracts representing single-stream port constraints, whereas the **assg_grp** is used for contracts representing inter-port constraints. For the **assg** operator, if the <port-symbol> represents an outport, the signature of the <function-reference> must be:

```
int (*)(int, time_tp)
```

If it represents an inport, the signature must then be:

```
int (*)(int, time_tp, time_tp)
```

The arguments of these functions are the order of the message in the communication (the int argument), the time in which the send operation is performed (the first **time_tp** argument), and the time in which it arrived (the second **time_tp** argument for functions monitoring inports). The function returns a positive value when the temporal properties of a message complies with the contract restrictions, and a non-positive value, otherwise. Messages that do not comply with the constraints are sent to designated exception handler ports.

For the **assg_grp** operator, if the <contract-id> specifies inter-inports constraints, the signature of the <function-reference> must be:

```
int (*)(int, int, time_tp)
```

If it specifies inter-outports constraints, the signature must then be:

```
int (*)(int, int, time_tp, time_tp)
```

The only difference between the signature of these functions and the functions monitoring non inter-port constraints is the additional first int argument, that identifies the sending port among the inter-related ones. Actually, it is an element of the enumeration that represents the group of inter-related ports.

B.1.3 Real-time Language Constructs

The general form for the specification of a real-time block is as follows:

```
<real-time-block> ::=
   within (<timing-constraint-expression>)
   do { <timed-block> }
   [responsiveness {<timed-block>}]
   [deadline {<timed-block>}]
   until (<condition>);
```

The semantics are as follows. Initially, the runtime system is called to decide whether the current system load allows the absorption of the real-time execution of the **<timed-block>** without violating the timing constraints specified in **<timing-constraints-expression>**. If the runtime is unable to accept it, control is passed to the statement following the **<real-time-block>**. Otherwise, the process enters the real-time mode, in which the **<timed-block>** is executed according to **<timing-con-**

straints-expression>, until **<condition>** evaluates to a positive value. Whenever control reaches the statement following the real-time block, a global variable has already been set by the runtime system to indicate the ending state of the real-time execution. It indicates if the runtime was unable to absorb the real-time execution in the first place (with a value of zero), or if the real-time execution was actually performed. In the last case, the value of the variable indicates how many times the **<timed-block>** was executed.

The behavior of the **<real-time-block>** is defined as follows:

```
<timing-constraints-expression> ::= [hard | soft] <schedule>
<schedule> ::=
    periodic  <period>[<deadline>][<start>][<responsiveness>]
    sporadic  <event>  <period>[<deadline>][<start>][<responsiveness>]
    aperiodic  <event>  <deadline> [<start>][<responsiveness>]
<period> ::= period <time-expression>
<deadline> ::= deadline <time-expression>
<start> ::= after  (<event-list>)
<event-list> ::= <event>  [OR  <event-list>]
<event> ::= select(<port>) | atEvent(<time-event>)
<time-event> ::= <time-expression>
<responsiveness> ::= response  <time-expression>
```

When the **<start>** clause is specified, the **<timed-block>** is activated for the first time only after one of the events defined in this clause occurs. When not specified, the block may be activated for the first time at any time. The events recognized by QuAL are the arrival of a message, which can be tested through the Concert/C `select` operator, and a time event, expressed after the keyword **atEvent**. The **<responsiveness>** clause indicates the maximum delay tolerated between the arrival of an event and the actual activation of the task (only for soft deadlines). Periodic and sporadic processes may be activated: (1) at a specific period of time after the last activation, expressed by a **<period>** clause, or (2) upon an external event such as the arrival of a message, expressed by an **<event>** clause. A strict deadline for finishing the current execution of the block is expressed by the **<deadline>** clause. When not specified, the default deadline is the time in which the next activation should begin. It is important to note the finer granularity for the expression of real-time constraints in QuAL, when compared to other real-time languages [halang91]. Constraints are expressed per block, rather than per process. Therefore, a QuAL process may have several blocks of code each with a different behavior. The process may execute a block of code that has a periodic behavior, and later on another that has a sporadic behavior.

The general form for a **<timed-block>** is as follows:

```
<timed-block>  ::= {<timed-instruction> | <timeout-block>}*
```

```
<timeout-block>  ::= timeout  ([<time-expression>])
                      { <instructions>* }
                      [expired  {<timed-instructions>*}]
```

A **<timed-block>** consists of a sequence of **<timed-instruction>**s and **<timeout-block>**s. A **<timed-instruction>** is any instruction for which the computational cost can be determined at compile time. An instruction or sequence of instructions for which the computational cost cannot be estimated has to be inside a **<timeout-block>**. At compile-time, the computational cost of these instructions is estimated to be equal to the **<time-expression>** given. If after this period of time the execution does not terminate, control is passed to the **<timed-instructions>** following the keyword **expired**, that will handle the exception. The exception handling mechanism is further explained in Section B.3. When the **<time-expression>** of a **<timeout-block>** is not specified, the compiler estimates (infers) a time value for the execution of the block based on previous executions of the same type of instructions.

When a process is executing in hard real-time mode, processing resources are reserved and scheduleability analysis performed based on a worst case performance. No exception messages are expect to be raised due to violations of the timing constraints expressed in the **<timing-constraints-expression>**. A process executing in hard real-time mode blocks and waits to be scheduled by the run-time system. It then executes the **<timed-block>** and the **<condition>** following the keyword **until** is evaluated. If it evaluates to a true value, control is passed to the statement following the **<real-time-block>**, and the process leaves the real-time mode of execution. If the condition evaluates to false, the process remains in the real-time mode and blocks waiting to be scheduled again. The runtime system only allows the process to engage the hard real-time mode of execution if enough resources can be reserved for the process to execute in its highest demanding mode (e.g., when sporadic tasks execute at every frame interval, behaving as periodic tasks).

When a process is executing in soft real-time mode, processing resources are reserved and scheduleability analysis performed predicting a non worst-case performance. Therefore, timing violations are likely to occur during transient periods of overload and exceptions will be raised upon timing violations. QuAL exception handling mechanism is designed to ease graceful recovery and degradation from such events. By providing both hard and soft real-time modes of execution, QuAL provides a trade-off between predictability and throughput. Since hard real-time mode of execution requires a worst-case performance scheduleability analysis, resources may be wasted during non overloaded periods. Processes executing in soft real-time mode can make use of these resources to execute, but they might have their timing constraints violated if the processes executing in hard mode need them.

B.2 Real-time Ports

The addition of the keyword **realtm** to a port declaration classifies the port as a real-time port, triggering the capture of the temporal properties of its messages and enabling a temporal-behavior-based retrieval of messages. Real-time ports support all the operations defined for Concert/C ports, plus the operations described in this section. When using Concert/C operators to access real-time ports, the messages temporal behavior are discarded and messages are retrieved in the order of arrival.

QuAL introduces the following extensions to the Concert/C receive and accept operators in order to provide access to the temporal behavior of messages:

```
receive_tm(<port-value>, <message-ref>,
           <time-ref>, <time-ref>, <time-ref>)
accept_tm(<port-value-list>,
           <time-ref>, <time-ref>, <time-ref>)

<time-ref> ::= <assignment-expression>
<port-value-list> ::= {<port-value>}+
```

The operator **receive_tm** causes a message to be dequeued from the port designated by <port-value>, and be placed on the storage designated by <message-ref>. It also places the message sending time, arrival time, and retrieval time on the storage designated by the first, second, and third **<time-ref>** arguments, respectively. If no message is present, it blocks waiting for the next message. The operator **accept_tm** works similarly to the Concert/C operator accept. That is, its function is to handle functionports, as explained in Section 2.4. Functionports are functions that can be called remotely. Thus, messages to this port are of type callmessages and represent a remote function call. **accept_tm** causes a callmessage to be dequeued from a functionport in the <port-value-list>, places its temporal behavior on the storage designated by the **<time-ref>** arguments, and invokes the function associated with the port, passing as parameters the values supplied in the message. If more than one functionport in the <port-value-list> has messages still to be processed, a port is chosen based on the Concert/C concept of fairness [auerbach92]. If no message is enqueued in any of the ports, the operation blocks until a message for any of the ports arrives.

QuAL provides an extensive set of operators to retrieve real-time port messages based on their arrival time:

```
rtm_receive(<time-ref>, <time-ref>, <port-value>, <message-ref>)
rtm_accept(<time-ref>, <time-ref>, <port-value-list>)
rtm_receive_tm(<time-ref>, <time-ref>, <port-value>, <message-ref>,
                    <time-ref>, <time-ref>, <time-ref>)
```

```
rtm_accept_tm(<time-ref>, <time-ref>, <port-value-list>,
                         <time-ref>, <time-ref>, <time-ref>)
```

The operator **rtm_receive** causes a message that arrived in the time interval defined by the first and second arguments to be dequeued from the port designated by <port-value> and be placed on the storage designated by <message-ref>. The operator **rtm_accept** dequeues and handles one call from any of the ports in the <port-value-list> that arrived in the time interval specified by the first two arguments. The operators **rtm_receive_tm** and **rtm_accept_tm** operate similarly, but, in addition, they place the message's sending time, arrival time, and retrieval time on the storage designated by the last three **<time-ref>** arguments, respectively. If there are no messages that arrived in the time interval specified, all these operations will block until either a message arrives or the time constraints are unreachable. In the last case, the operations will return an error value, signaling the exception.

Similar extensions of the Concert/C select and poll operators are also provided:

```
<int> rtm_select(<time-ref>, <time-ref>, <port-list>)
<int> rtm_poll(<time-ref>, <time-ref>, <port-list>)
```

The operator **rtm_select** waits (if necessary) until one of the inports in the <port-list> has a message that arrived in the interval of time defined by the first and second arguments or the time constraints become unreachable. If one or more inports have a message which satisfies the restriction, it returns the index of one of these inports. Operator **rtm_poll** operates similarly, except when no ports in the list have messages satisfying the restrictions in which case it returns 0 immediately (that is, it never blocks).

B.3 Specification of QoS and Processing Control

The exception handler ports associated with a port are specified through the **<handlers>** clause as follows:

```
<handlers> :: = [handlers {<list-of-handlers>}]
<list-of-handlers> ::= <net-hndlr> <app-hndlr>
<net-hndlr> ::= net_handler <port-reference>
<app-handler> ::= [app_handler <port-reference>;]
   [grp_app_handler <port-reference>;]
```

If either the application or the underlying system violates the network-level QoS constraints, an exception message is sent to the inport specified in the **<net-hndlr>** clause. When the QuAL monitoring mechanism reaches the conclusion that a message violates the single-stream application-level QoS constraint being complied by the port, an exception message is sent to the inport specified after the keyword **app_hndlr**. Similarly, a message is sent to the port specified after the keyword **grp_app_hndlr** if a message violates the inter-port constraint being complied by the port. The exception message includes the message that originated the exception. If the exception was detected at the time the message was sent, the message does not proceed to reach the receiving end. If the exception was detected at the time the message arrived at the receiving end, the message does not get enqueued in the respective inport.

QuAL also provides an exception handling mechanism for handling timing constraints violations of soft tasks. Either immediately before or after the activation of a soft `<timed-block>`, control may be passed to an exception handler. This happens whenever a timing constraint has been violated causing an exception to be raised. The following timing constraints may be violated: (1) responsiveness—the system is unable to schedule the process within the response time required and (2) deadline—the system was unable to provide enough processing resources for the process to finish execution before its deadline. The general form for the specification of a real-time block is described below. The handlers for these exceptions are expressed after the **responsiveness** and **deadline** keywords, respectively.

```
<real-time-block> ::=
   within  (<timing-constraint-expression>)
   do  {  <timed-block>  }
   [responsiveness  {<timed-block>}]
   [deadline  {<timed-block>}]
   until  (<condition>);
```

QuAL also provides an exception handling mechanism for handling expiration of time-out expressions. A timeout block may be followed by a handler block that is executed whenever the timeout expires. The general form for the specification of a timeout block is as follows:

```
<timeout-block>  ::= timeout  (<time-expression>)
                      { <instructions>*  }
                      [expired  {<timed-instructions>*}]
```

B.4 Application Management Automation

QuAL extends the instruction set of a base language to support the SNMP get operations and *monitoring instructions*. The SNMP get operations allow the retrieval of data from a QoS-MIB. A monitoring instruction enables the specification of a trap condition to be monitored by the QuAL SNMP agent.

The general form for the specification of a SNMP get operation is as follows:

```
<get-operation> ::= get(<OBJECTIDENTIFIER>)
```

where **<OBJECTIDENTIFIER>** uniquely identifies a field in a QoS-MIB according to the SMI standard.

The general form for the specification of a monitoring instruction is as follows:

```
<monitoring-instruction>  ::= monitor_cnd(<management-cond-expr)
```

```
                                    mgt_handler <port-reference>;
  <management-cond-expr>  ::= <management-cond>
                                    [<logic-operator>  <management-cond-expr>]
  <management-cond> ::= <OBJECTIDENTIFIER>  <comparative-operator>
                             <constant-expression>  |  <OBJECTIDENTIFIER>
```

Each **<monitoring-instruction>** consists of a **<management-cond-expr>** and an exception handler port. The **<management-cond-expr>** is a boolean expression whose operands are either constant values, or access operations to the application's MIBs. Therefore, its value changes only when one of the MIBs referred in the expression is updated. When this expression evaluates to a true value, the event being monitored has occurred and must be handled. A message is then sent to the exception handler port specified after the keyword **mgt_handler**.

At compile time, the dependencies between these boolean expressions and the MIBs are inferred. During run time, the execution of a monitoring instruction causes these dependencies, along with the boolean expressions and handler ports, to be forwarded to the QuAL SNMP agent. During run time, every time a MIB is updated, the dependent **<management-condition>**s are re-evaluated to reflect the update. Whenever they evaluate to a true value, a message is sent to the designated exception handler port, providing a real-time monitoring of the MIBs in an application-oriented manner.

C QuAL Communication-Support System

The connection establishment process in QuAL is depicted in Figure C.1. Actions are described inside squares. The dotted horizontal lines divide the layers in which the actions take place and the dotted arrows show the action flow between layers. The bold horizontal line represents the time and the vertical arrows describe important events in the time dimension. The big dots represent the repetition of the action on its left at every intermediate node in the path from the sender to the receiver, or vice-versa. As discussed in previous sections, the port data type is extended to include the specification of QoS attributes and intervals for their QoS deviation bound values. Furthermore, exception handler ports may also be defined, specifying where messages signaling exceptions should be sent. All this happens during the application development time. Only ports with *compatible* QoS attributes can be bound, that is, ports with a non-null intersection of their QoS deviation bound intervals for each attribute specified. This intersection interval is calculated during binding-time. Two bound ports can only communicate if the runtime is capable of establishing a connection whose QoS values belong to the respective intersection interval. The QuAL runtime is responsible for choosing the transport protocol that best supports the QoS required for the communication, for mapping the abstract QoS attributes and QoS deviation bound values into transport-layer specific parameters, and for opening the respective connection. To open a connection, the QuAL runtime interacts with transport-layer entities requesting the opening of a connection with the respective QoS parameters.

The connection establishment process is specific to a transport protocol. In the case of the ST-II transport protocol for instance, the ST-II entity at the origin performs three main tasks: (1) it allocates local processing and buffering resources, (2) it updates the current QoS parameters of the communication to reflect the cost of the resources allocated for the communication and the delay overhead added by the processing that will be performed at that node, and (3) it forwards the updated request to the next node in the path to the destination. Intermediate nodes in the path to the destination behave in a similar fashion until the request reaches the ultimate target. The ST-II entity at the target node also allocates resources and updates the QoS parameters of the communication. Then it notifies the local QuAL runtime of the connection establishment request as well as of the current QoS parameters of the communication. Examples of QoS parameters are the accumulative end-to-end delay and the least amount of processing and buffering resources allocated by the nodes in the path. This amount dictates the maximum message rate supported on the particular connection. If the given transport-layer QoS parameters are translated into QuAL QoS attribute values that belong to the intersection interval of the ports being connected, the connection establishment is confirmed by the QuAL runtime. The QuAL runtime sends an acknowledgment to the transport-layer entity at the target node and this entity propagates the information backwards to the intermediate nodes in the path. Each node updates the status of the connection and may release resources if the final QoS of the connection requires less resources than the ones initially reserved. At the source node, the transport-layer entity notifies the QuAL runtime of the connection establishment as well as of the final communication QoS values.

After a connection is opened, the runtime creates Network-level CMPs (NCMPs) at both ends that monitor the QoS of the communication. Figure C.2 depicts the monitoring process after the connection is established. The communication illustrated is over an ST-II connection, but the semantics are the same regardless of the transport protocol used. The dashed horizontal line divides the layers in which the monitoring is performed (runtime layer and ST-II layer), and the layer that uses the monitoring services (the application layer). NCMPs are represented by the cylinder shaped objects, whereas the connection monitoring entities at the transport-layer are represented by oval circles. These objects also describe the main task performed by these entities. At the application-layer, the straight arrows represent application-level data flow, whereas the dashed ones represent information sent from the NCMPs to the applications. At the ST-II layer, the straight arrows represent the flow of the application-level data and of the NCMPs control information, whereas the dashed arrows represent the flow of information specific to the transport protocol. The NCMPs prevent the application from misusing the connection resources. NCMPs also interact with the transport-layer entities to transmit data and monitor the connection. They add QuAL control information to the user data being transmitted, such as an index to identify the message and the sending and arrival time of the message. This information is used to perform measurements of the communication QoS, such as loss rate and end-to-end delay, and is also made accessible to the applications. The timing of the message can be used by the application to further monitor and study the communication temporal behavior or to prioritize message service based on their timing. NCMPs also process control information received from the transport-layer entities, such as the detection of a link failure or a notification that a node has to release some of the resources previously allocated. NCMPs try to recover immediately by opening another connection or re-negotiating the connection QoS, for instance. When they are unable to recover, NCMPs map the events that occurred into exception messages and send them to the designated exception handler ports. Exceptions detected by the NCMP at the origin are propagated to the NCMP at the target node, and vice-versa. At the application-layer, applications are exposed only to exception messages and are free to decide how to handle them.

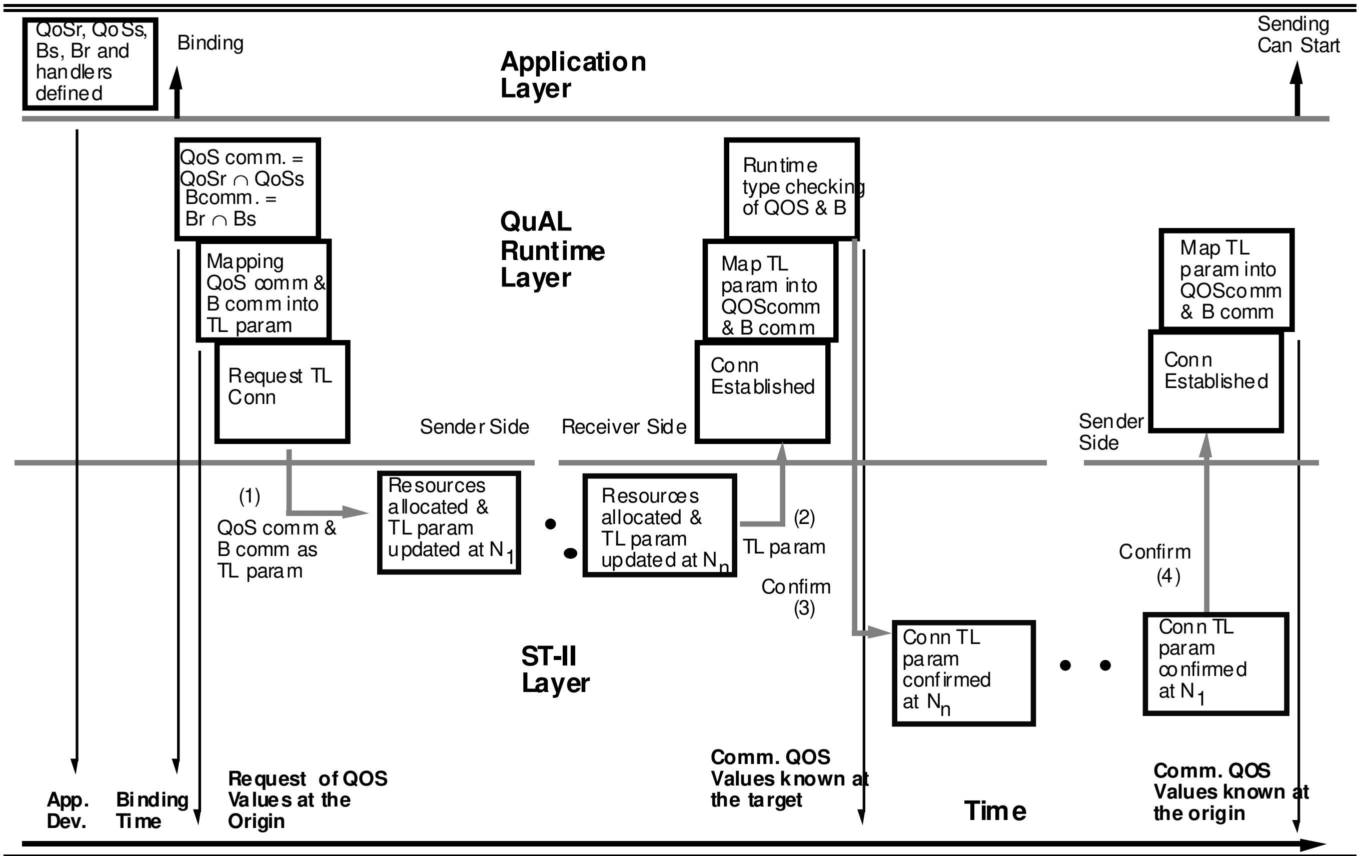

Figure C.1: Communication Establishment in QuAL

QuAL also provides operators for the dynamic re-negotiation of QoS parameters. After a binding has been performed, the process owning the binding or the input port can request the re-negotiation of the QoS parameters. The new QoS values must belong to the QoS intersection interval of the ports connected. The request is forwarded to the local NCMP that will interact with the respective transport-layer entity, similarly to when the connection was initially established. If the transport protocol being used allows the re-negotiation of QoS values on a connection already established, this is the approach used. Otherwise, a new connection is opened. All this activity will take place concurrently with data transmission on the connection previously opened. Only if and after the new QoS are acquired, the NCMPs will start sending and monitoring according to the new QoS values.

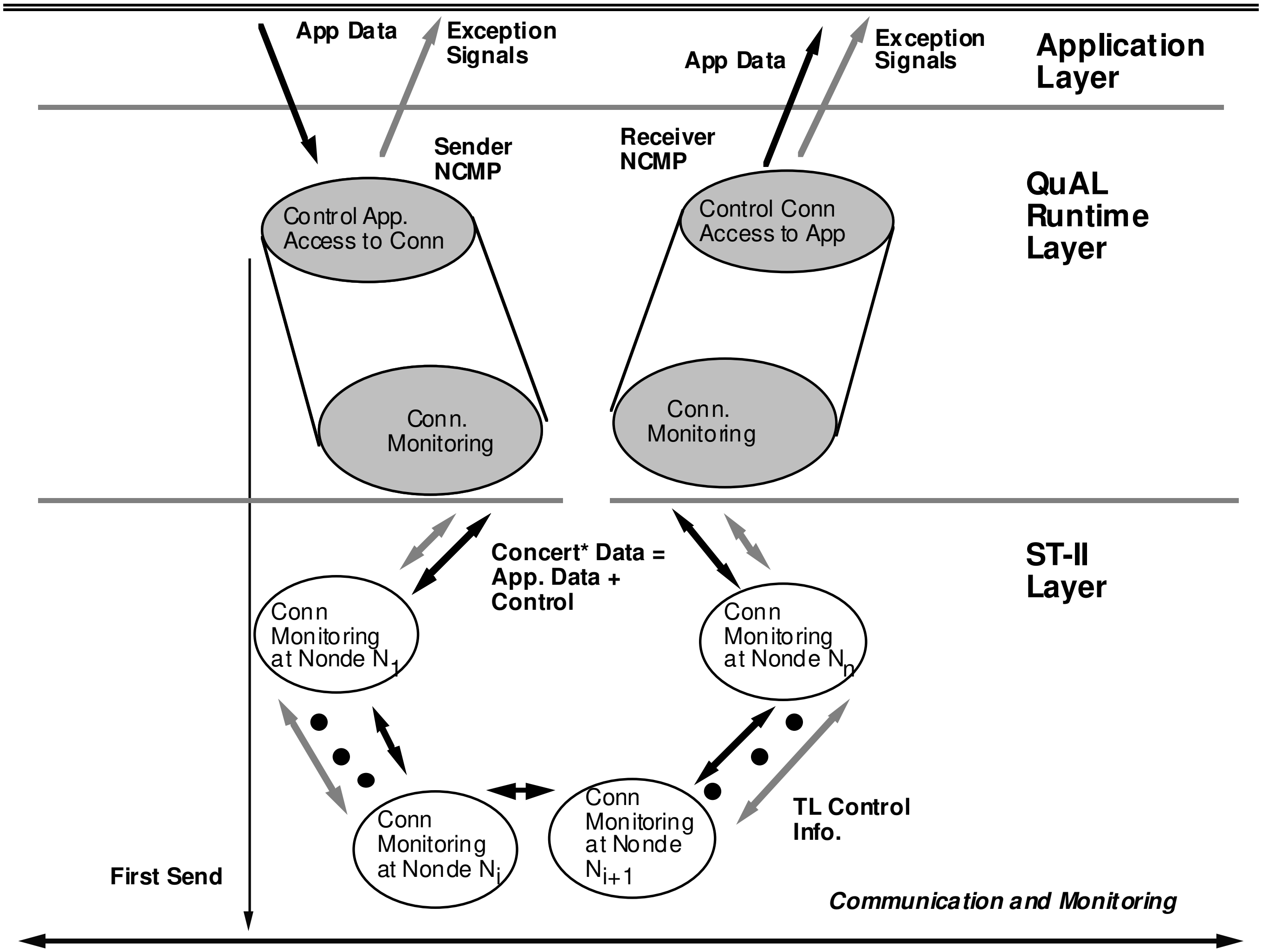

Figure C.2: Communication Monitoring in QuAL.

In summary, NCMPs interact with transport-layer entities to negotiate QoS, transmit data, and process control information sent by these entities. QuAL runtime is designed to interact with diverse transport protocols that may provide the QoS needed by the application. Control information sent by transport-layer entities is filtered by the NCMPs and translated into exception messages whenever necessary. QuAL thus shelters heterogeneity for QoS negotiation and monitoring by providing an uniform abstraction of exception messages to applications.

D QuAL Overall Architecture

In this section, we first describe the architecture of the QuAL runtime system and identify its distinctive aspects. We then proceed to explain in grater detail the QoS monitoring process and how we handle the real-time scheduling of processes.

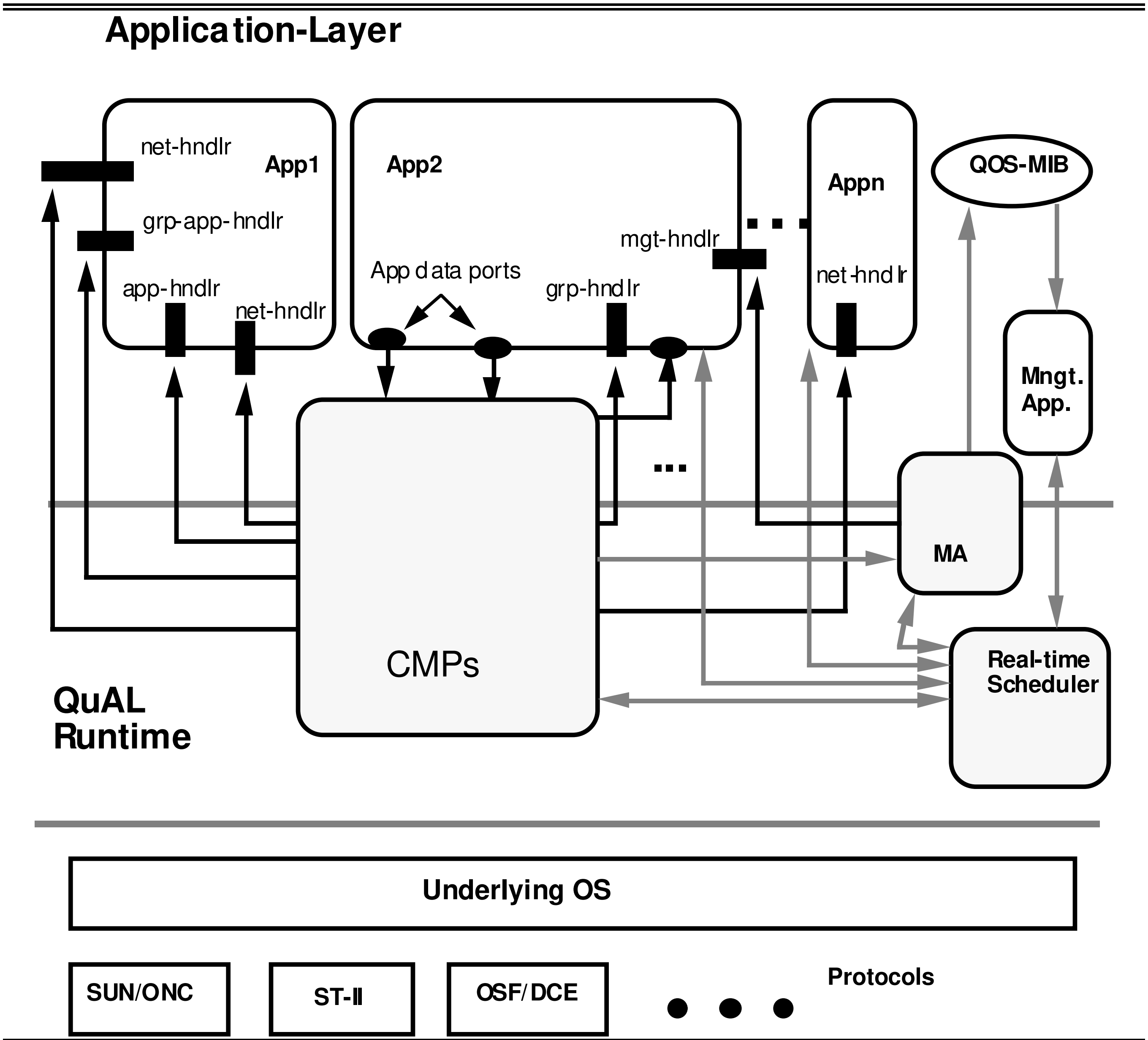

Figure D.1: QuAL Runtime Architecture

The architecture of the QuAL runtime for one particular machine and its interaction with other layers is depicted in Figure D.1. The thick horizontal lines delimit the application-layer, the runtime, and the other layers that provide services to the runtime. At the application-layer, QuAL processes and management applications are represented by rectangles. Ports that communicate application-level data are represented by small oval circles, whereas ports communicating exception messages sent by the runtime are represented by small rectangles. The straight arrows show the information flow. The runtime consists of three main components: the CMPs' module, the real-time scheduler, and the Management Agent (MA). CMPs monitor at the communication-level of abstraction. The real-time scheduler is responsible for managing the processing resources and for scheduling the activities accordingly. The dashed arrows show the interactions between the real-time scheduler and the several processes and modules depicted. The MA accesses and manages the QoS-MIB. The QuAL runtime layer supports the application layer and serves as an interface to both the OS and network layers of an underlying system. Furthermore, it interacts with management applications through MIBs providing feedback on the performance of the applications' communication activities.

The CMPs are responsible for assuring and monitoring QoS attributes, for notifying respective applications in case any violation occurs, and for providing monitoring information to the MA. The CMPs interact with applications through data ports and exception handler ports. CMPs receive and deliver application-level data to applications through data ports and report on any QoS violation through exception handler ports. Part of the CMPs' module is general and common to all applications and it is responsible for monitoring the interaction and services provided by the protocol-layer components. This part consists of NCMPs and it was described in Section C of the appendix. The other part of the CMPs' module can be customized by applications to control and monitor communications in an application-dependent manner, thus accessible at the application-layer. This part implements the monitoring of application-dependent QoS parameters. An application-dependent QoS monitoring might monitor, for instance, the sending rate of messages, independently of the services provided by the communication underneath, and cause the reduction of the transmission rate by discarding every other message sent by an application. The CMPs may also interact with system management applications by providing them with the information collected during the communication monitoring process. Upon request by application developers, monitoring information is sent to the MA that is responsible for storing this information in the QoS-MIBs.

The real-time scheduler manages the processing resources provided by the underlying OS, scheduling all processing activities based on their timing constraints, such as proximity of their deadlines. Any processing may occur either in real-time or in non real-time mode. Real-time mode processing has priority over non-real-time mode. Before entering the real-time mode, a processing activity must first consult the real-time scheduler regarding the viability of allocating and scheduling its processing according to its timing constraints. The real-time scheduler may accept or reject the request. If it accepts, the real-time scheduler will schedule the process to comply with its timing constraints. If it rejects, the process continues executing in non real-time mode and it is scheduled only when there are no real-time process ready to execute. It is important to note that all modules are scheduled through the real-time scheduler, including the CMPs' module. Thus, before a QoS dependable communication is opened, the real-time scheduler is consulted to determine whether it is possible to execute the CMPs needed by the communication in a real-time fashion.

The MA implements an SNMP agent and provides the following functionalities:

- Stores the data generated by the CMPs in the QoS-MIBs;
- Accesses the QoS-MIBs in response to requests of applications to retrieve data, and
- Monitors the data in the QoS-MIBs. That is, it maintains the dependencies between the trap conditions and the tables in the QoS-MIBs, it evaluates the conditions when the respective tables are updated, and sends trap messages to designated exception handler ports, whenever these conditions evaluate to true.

A MA is accessible at the application-layer through the definition of trap conditions.

The following sections further elaborate on the CMPs' module and on the real-time scheduler.

D.1 The CMPs' Module

QuAL recognizes two classes of QoS parameters. They are network-level and application-level QoS parameters. The monitoring and control of network-level QoS parameters were detailed in Section C of the appendix. They interface applications' access to network-layer services. Therefore, for each connection established, there is a corresponding NCMP at the sending and at the receiving sides. Application-level QoS parameters are handled in a similar way, but it comprises to interfacing applications' access to NCMPs. The main goal is to monitor the communication in an application-dependent manner.

The interaction between application processes and runtime system processes is illustrated in Figure D.2. The figure shows an example of an application's configuration. Application processes are represented by white rectangles and runtime processes are represented by the shaded ones. The straight arrows show the flow of application-level data sent by application-level processes and of exception messages sent by the runtime to application-level processes. The other arrows show either interactions between the real-time scheduler and the processes depicted, or management information flow. Several configurations for the monitoring of QoS are illustrated in this figure. A message sent through the left most port of application **`App2`**, is first sent to an Application CMP (ACMP) that checks the single-stream QoS attributes of the communication. The ACMP decides whether the message should reach the next step in the communication or be forwarded to an exception port (represented in the picture by a port referenced as `app-hndlr` port). Messages that proceed are sent to their respective Group CMP (GACMP) that checks the group QoS attributes of the communication. Group QoS attributes of a communication checks the QoS of a message in relation to the traffic behavior of other ports in the group. In this case, the second left most port of application **`App2`**. The GACMP either sends the message to the NCMP monitoring the network-level QoS attributes of the connection or to an exception handler port (represented in the picture by a port referenced as `grp-app-hndlr`). The monitoring performed by a NCMP constitutes the last stage of the communication monitoring. Messages that pass this stage are delivered to the transport-layer entities. Messages that violate the network-level QoS attributes are forwarded to an exception handler port (represented in the picture by a port referenced as `net-hndlr`). As illustrated in this example, not all application-level ports have such a tight control. Ports may have only group QoS monitoring and no single-stream monitoring (as the second left most port of **`App2`**), or vice-versa. Furthermore, ports may have no application-level QoS monitoring and be monitored only by the network-level QoS parameters (as the right most port of **`App2`**). If a program is compiled with a management option, the CMPs report on the monitoring they are performing to the MA. The MA is responsible for updating the QoS-MIB with this information and for sending trap messages to a designated exception handler port (represented in the picture by a port referenced as `mgt-hndlr`), upon detection of exceptional conditions. These conditions are defined by the application-developer.

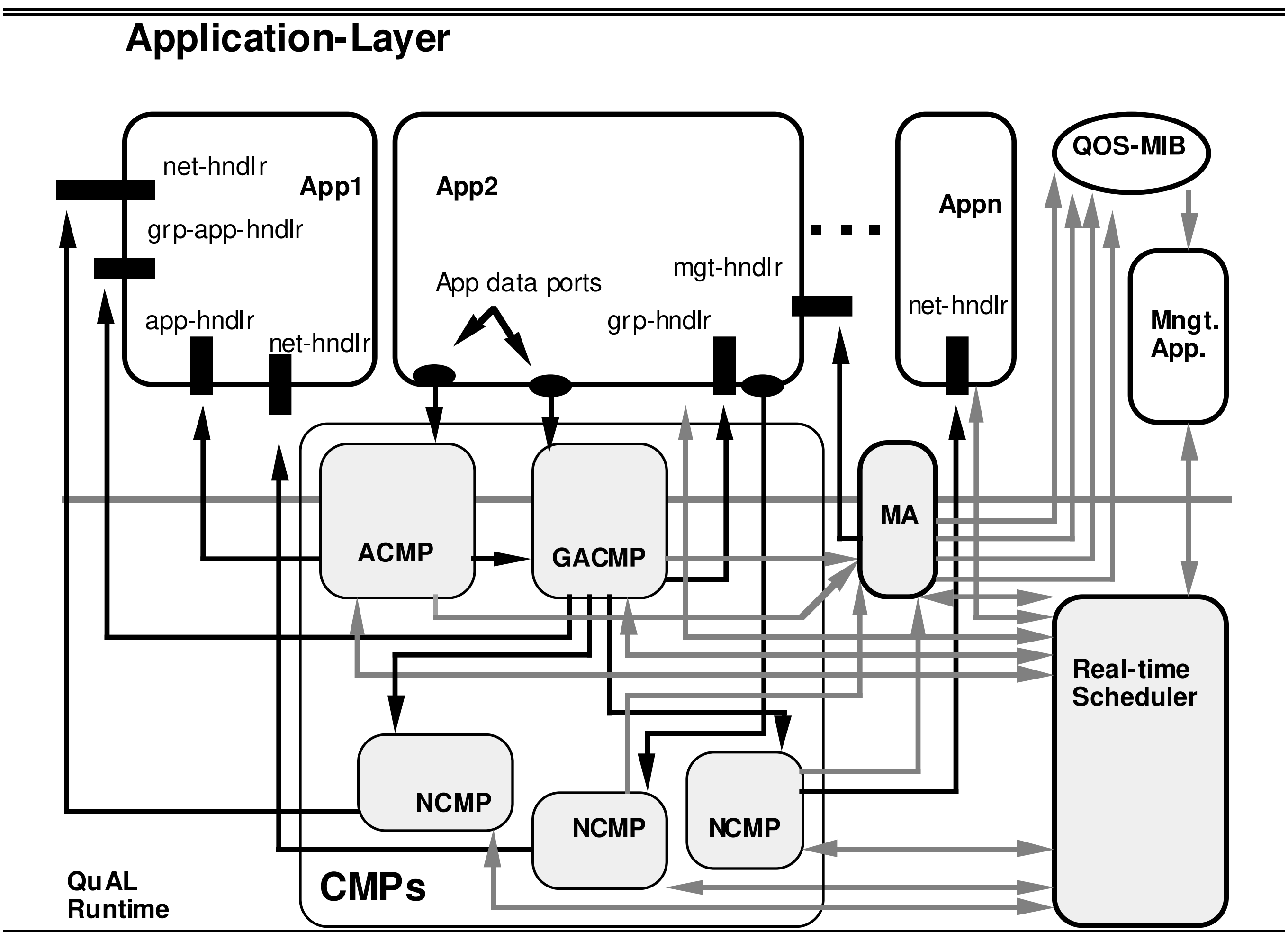

Figure D.2: Example of a CMPs' Module Configuration

D.2 The Real-time Scheduler

Monitoring and management of QoS attributes (e.g., end-to-end delay and transmission rate) can only be effective if performed in real-time and if the applications are powerful enough to express the timing constraints associated with their execution. Therefore, QuAL includes support for real-time language constructs for applications' timing constraints specification. These constructs are compiled into interactions with the real-time scheduler. Before entering the real-time mode of execution, an application consults the real-time scheduler regarding the possibility of allocating processing resources for its execution complying with its timing constraints. The real-time scheduler manages the overall system load and is responsible for allocating resources accordingly. Real-time processes can execute either in the hard or soft mode. The real-time scheduler allocates resources for processes executing in hard real-time mode based on the worst-case performance cost of their executions and processes are guaranteed to have their timing constraints preserved. The allocation of resources for processes executing in the soft real-time mode is not so tight. It is assumed that the processes will not execute in their worst-case performance mode. Nevertheless, the real-time scheduler uses a best effort approach in scheduling the soft processes according to their timing constraints. If the constraints are violated, QuAL provides an exception handling mechanism for the graceful recovery or degradation from the transient period of overload. To avoid a chaos situation in which the timing constraints of the processes executing in the soft mode are always violated, the scheduler monitors the system load and only accepts new processes to execute in real-time if the load does not exceed a certain threshold.

QuAL can be implemented directly on top of several real-time OS (e.g., Split-Level Scheduler [govindam91]). However, most of these OS are not of general purpose or not widely available. Other OS (e.g., Mach) provide only some of the real-time capabilities needed by QuAL (e.g., kernel-level threads and time-bounded system calls), failing to support all the features needed (e.g., scheduling of tasks based on their timing constraints). The main goal of the QuAL real-time scheduler is to bridge the gap between the features needed by QuAL and the features provided by the underlying OS. For our first prototype, we needed an underlying environment capable of supporting an existing implementation of Concert/C and ST-II, being the UNIX 4.3 OS our only choice. UNIX 4.3 lacks basic features required by QuAL from an underlying OS (e.g., kernel-level threads and time-limit for the release of shared resources by the OS). However, as a proof of concept, we are simulating a real-time scheduler for QuAL on top of UNIX 4.3. We assume that QuAL applications are the only applications currently running and we consider in the scheduleability analysis the overhead introduced by trying to overcome all the existing deficiencies.

The real-time scheduler we are prototyping analyzes the system load and reserves resources for hard real-time processes based on results described in [jeffay91]. In [jeffay91], a *task* is defined as a sequential process that is invoked by each occurrence of a particular *event*. An event is a stimulus generated by a process that is either external to the system (e.g., interrupt from a device) or internal to the system (e.g., clock ticks or the arrival of a message from another process). It is assumed that events are generated repeatedly with some maximum frequency; thus the time between successive invocations of a task will be of some minimal length. Each invocation of a task results in a single execution of the task at a time specified by a scheduling algorithm. Formally, a task is a pair (c, p) where c is the computational cost, and p is the *period*, that is, the minimal interval between invocations of the task. The clock ticks are discrete events and c and p are expressed as multiples of the interval between clock ticks. Two paradigms of task invocation are considered: *periodic*, where p specifies the constant interval between invocations, and *sporadic*, where p specifies the minimum interval between invocations. Results published in [jeffay91] show that the scheduleability of a set of sporadic and periodic tasks can be efficiently determined. More precisely, given that the cost and the period of the tasks are known, the scheduleability can be determined in linear time and the system load can also be given an upper bound. These results are explored in QuAL in the following way. A QuAL process starts executing in real-time mode when a real-time language construct is executed. The real-time language construct requires the specification of the period p of the code sequence that will execute in real-time, the event that triggers the execution, and a termination condition, upon which the process should leave the real-time mode. During compilation-time, a worst case performance analysis is made to calculate the cost of this code sequence. QuAL requires the specification of a time-out period for every instruction or block of instructions for which the cost cannot be calculated by the compiler. The compiler translates a real-time block into interactions with the QuAL real-time scheduler. The real-time scheduler is then responsible for reserving resources and scheduling processes according to their timing requirements.

QuAL supports soft and hard real-time mode of execution. An example of an application that has hard constraints is the monitoring of a nuclear plant. If preventive actions are not taken a specific amount of time after dangerous events are triggered, the system may be destroyed. However, resources are allocated in a preventive way, based on worst case performance load, and during low-load periods, the system is underutilized. Systems supporting soft real-time applications trade a higher throughput for eventual violations of timing constraints. An example of soft real-time applications is a video application in which missing video frames is acceptable under some circumstances. QuAL exception handling mechanism provides graceful degradation and recovery from periods of transient overload.

Purely real-time-application-driven languages have no constructs that can take arbitrarily long to execute. To enable the calculation of its computational cost, a task is required to satisfy the following requirements:

- *No recursion or cycles in the function call chain:* the system is unable to calculate an upper bound on the depth of the recursion. This makes it impossible to determine at compile time: (1) the worst execution time; (2) the size of the heap needed to store all the activation records.
- *No hierarchical memory access:* Remote memory access over the network may take an unbounded amount of time.

- *No dynamic memory allocation:* The system cannot guarantee, during compile time, that there will be enough memory to execute the program or if a garbage collection may be needed, that can take an unpredictable amount of time.
- *No blocking or event-driven statements:* The scheduler would have to reserve the CPU for the entire period in which the event may happen. Blocking statements include I/O, and event-driven statements include the arrival of a message.
- *No contention of resources:* The system cannot calculate how long it will take for a shared resource to be freed.
- *All loops must be bounded:* to allow a worst case execution time estimation.

To avoid the enforcement of these restrictions in extending a general purpose language such as C, QuAL provides the time-out mechanism. That is, if the compiler cannot calculate statically the worst-case cost of an instruction or sequence of instructions, either the application developer or the compiler calculate an approximation of the cost for the execution of the code. The runtime system raises an exception whenever the execution does not end in the time period predicted. This mechanism is more flexible and of finer granularity.

The real-time scheduler schedules processes based on a slight variation of the *Earliest Deadline First* (EDF) algorithm [liu73], that recognizes processes executing in hard or soft mode. When selecting a process for execution, an EDF scheduling algorithm chooses the task with an uncompleted invocation with the earliest deadline. Ties between tasks with identical deadlines are broken arbitrarily. A task executes to completion or is pre-emptied by the scheduler if another task with an earlier deadline is invoked. The QuAL real-time scheduler uses pure EDF for processes executing in hard real-time mode. When there is no hard real-time process executing, the real-time scheduler uses the EDF to schedule soft real-time processes. Soft real-time processes are pre-emptied by hard real-time processes. Processes executing in non-real-time mode are only executed when there is not any real-time process, either hard or soft, ready for execution.

The universality of the EDF algorithm for scheduling sporadic and periodic tasks without preemption is also shown in [jeffay91]. This means that if any non-preemptive algorithm schedules a set of sporadic tasks, then the EDF algorithm will as well. The non-preemptive scheduling is a more restrictive case of the preemptive scheduling. Thus, this result also proves the universality of the EDF algorithm used in our case.

The real-time scheduler design follows the *split-level* scheduler architecture described in [govindan91]. The split-level scheduling is a scheduler implementation technique that minimizes interactions across different address spaces between the scheduler and the processes being scheduled, while correctly prioritizing Light-Weigh Processes (LWP) in different Heavy-Weigh Processes (HWP). Multiple LWPs inside a HWP share a *User-Level Scheduler* (ULS). The ULS monitors the execution of its LWP and interacts with a *Kernel-Level Scheduler* (KLS), that monitors all the ULSs. KLS and ULSs share information through shared memory mechanisms, regarding the LWP that has the highest priority and should execute next. The KLS schedules the ULS that monitors such LWP and sleeps until a LWP monitored by a different ULS gains priority. The ULS selected by the KLS schedules the respective LWP and performs any context switching needed, if another LWP in its address space gets priority. This scheme is general enough to allow any metric to be used in defining the notion of priority and in ordering them.

In QuAL, the real-time scheduler plays the role of the KLS. The QuAL real-time language constructs are designed to allow the implementation of the QuAL real-time scheduler on top of any OS that provides a real-time service access interface according to the POSIX standards [posix90]. The standard guarantees, among other features, that the OS kernel calls have a bounded worst case execution time. Therefore, even though the real-time scheduler is running on top of another OS, it can provide a guaranteed real-time responsiveness. This approach is less efficient than having QuAL executing on top of an embedded real-time OS that would perform the scheduling. However, current available technology provides enough processing power to make the use of this approach reasonable and this design choice allows the support for QuAL applications on top of several heterogeneous general-purpose platforms.

References

[ada83] *The Programming Language Ada Reference Manual*, Spring-Verlag, Berlin-Heidelberg-New York-Tokyo, 1983, ANSI/MILSTD-1815A.

[anderson90] Anderson, D.P., Tzou, S., Wahbe, R., Govindan, R., and Andrews, M., "Support for Continuous Media in the DASH System," in *Tenth International Conference on Distributed Computing Systems,* Paris, May 1990.

[andrews91] Andrews, G.R., "Paradigms for Process Interaction in Distributed Programs," *ACM Computing Surveys*, vol. 23, no. 1, 49-90, March 1991.

[auerbach91] Auerbach, J.S., Bacon, D.F., Goldberg, A.P., Goldszmidt, G., Kennedy, M.T., Lowry, A.R., Russel, J.R., Silverman, W., Strom, R.E., Yellin, D.M., and Yemini, S.A., "High-level language support for programming reliable distributed systems," in *First CASCON International Conference,* Toronto, Canada, October 1991.

[auerbach92] Auerbach, J., "Concert/C Specification," Tech. Rep., November 1992.

[bal89] Bal, H.E., Steiner, J.G., and Tanembaum, A.S., "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, vol. 21, no. 3, 261-322, September 1989.

[barnes76] Barnes, J., *RTL/2 Design and Philosophy*, Heyden, London, 1976.

[campbell93] Campbell, A., Coulson, G., Garcia, F., Hutchison, D. and Leopold, H., "Integrated Quality of Service for Multimedia Communications," in *IEEE INFOCOM,* 1993, San Francisco, March 1993.

[chesson88] Chesson, G., "XTP/PE Overview," *Proceedings 13th Conference on Local Computer Networks*, Pladisson Plaza Hotel, Minneappolis, Minnesota, pp 292-296, 1988.

[cohen81] Cohen, D., "A Network Voice Protocol NVP-II," Tech. Rep., April 1981.

[cole81] Cole, E., "PVP - A Packet Video Protocol," Tech. Rep., W-Note 28, August 1981.

[cordy81] Cordy, J. and Holt, R., "Specification of Concurrent Euclid," Tech. Rep., CSRG-133, August 1981.

[dce91] *OSF DCE*, Open Software Foundation, February, 1991, Release 1.0 Developer's kit documentation.

[deprycker93] De Prycker, M., Asynchronous Transfer Mode, Ellis Horwood, 1993.

[donner88] Donner, M. and Jameson, D., "Language and operating systems features for real-time programming," *Computing Systems* .

[fish89] Fish, R.S., "Cruiser: A Multi-media System for Social Browsing," *The ACM SIGGRAPH Video Review Suplement to Computer Graphics*, Vol 45, No 6, Videotape, 1989.

[florissi94] Florissi, P. G. S., "QoS-MIBs Design," Tech. Rep., Columbia University, January 94.

[forgie79] Forgie, J., "ST - A Proposed Internet Stream Protocol," Tech. Rep., IEN 119, September 1979.

[garey75] Garey, M. and Johnson, D., "Complexity results for multiprocessor scheduling under resource constraints," *SIAM Journal on Computing*, vol. 4, no. 4, 397 - 411, December 1975.

[govindan91] Govindan, R. and Anderson, D.P., "Scheduling and IPC mechanisms for continuous media," in *13th ACM Symposium on Operating Systems,* 1991.

[grimshaw89] Grimshaw, A., Silberman, A., and Liu, J., "Real-Time Mentat programming language and architecture," in *GLOBECOM 89,* 1989, pp. 1-7.

[halang91] Halang, W.A. and Stoyenko, A.D., *Constructing Predictable Real Time Systems*. Boston/Dordrecht/London: Kluwer Academic Publishers, 1991.

[henn75] Henn, R., "Deterministische Modelle für die Prozessorzuteilung in einer harten Realzeit-Umgebung," Ph.D. thesis, Technical University Munich, 1975.

[hoare78] Hoare, C.A.R., "Communicating Sequential Processes," *Communications of ACM*, vol. 21, no. 8, 666-677, August 1978.

[ishikawa90] Ishikawa, Y., Tokuda, H., and Mercer, C., "Object-oriented real-time language design: Constructs for timing constraints," Tech. Rep., CMU-CS-90-111, March 1990.

[iso] *Information processing systems - Open Systems Interconnection - Application Layer Structure* , International Standard ISO.

[jeffay91] Jeffay, K., Stanat, D.F., and Martel, C.U.M., "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks," in *Twelfth IEEE Real-time Systems Symposium,* San Antonio, Texas, December 1991, pp. 129 - 139.

[jeffay92] Jeffay, K., Stone, D.L., and Smith, F.D., "Transport and Delay Mechanism for multimedia Conference Across Packet-switched networks," *To Appear on Journal of Computer Networks and ISDN Systems*, .

[jul88] Jul, E., Levy, H., Hutchinson, N., and Black, A., "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, 109-133, February 1988.

[kap77] Kappatsch, A., "Full PEARL language description," Tech. Rep., KFK-PDV 130, 1977.

[keller93] Keller, R. and Effelsberg, W., "MCAM: An Application Layer Protocol for Movie Control, Access, and Management," in *First ACM International Conference on Multimedia,* Anaheim, 1993.

[kernighan88] Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*, second. Englewood Cliffs, NJ: Prentice Hall, 1988.

[kieburtz76] Kieburtz, R. and Hennessy, J., "TOMAL – A high-level programming language for microprocessor process control applications," *ACM SIGPLAN Notices*, vol. 11, no. 4, 127-134, April 1976.

[kirk70] Kirk, D., *Optimal Control Theory* Prentice-Hall, 1970.

[kligerman86] Kligerman, E. and Stoyenko, A., "Real-Time Euclid: A language for reliable real-time system," *IEEE Transactions on Software Engineering*, vol. 12, no. 9, 941 - 949, September 1986.

[kong90] Kong, M., Dineen, T.H., Leach, P.J., Martin, E.A., Mishkin, N.W., Pato, J.N., and Wyant, G.L., *Network Computing System Reference Manual*. NJ: Englewood Cliffs, Prentice-Hall, 1990.

[lee84] Lee, I., "A programming system for distributed real-time applications," in *IEEE Real-Time Systems Symposium,* December 1984, pp. 18-27.

[lee85] Lee, I. and Gehlot, V., "Language constructs for distributed real-time programming," in *IEEE Real-Time Systems Symposium,* 1985, pp. 57-66.

[leinbaugh80] Leinbaugh, D., "Guaranteed response times in hard-real-time environment," *IEEE Transactions on Software Engineering*, vol. 6, no. 1, 85 - 91, January 1980.

[lin87] Lin, K., Natarajan, S., and Liu, J., "Imprecise results: Utilizing partial computations in real-time systems," in *IEEE Real-Time Systems Symposium,* December 1987.

[lin88] Lin, K.J. and Natarjan, S., "Expressing and maintaining timing constraints in FLEX," in *IEEE Real-time Systems Symposium,* December 1988.

[liu73] Liu, C. and Layland, J., "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, 46 - 61, 1973.

[mok78] Mok, A. and Dertouzos, M., "Multiprocessor scheduling in a hard-real-time environment," in *Seventh Texas Conference on Computing Systems,* November 1978, pp. 5.1 - 5.12.

[nageli79] Nägeli, H. and Gorrengourt, A., “Programming in PORTAL: An introduction,” Tech. Rep., 1979.

[nelson81] Nelson, B. J., “Remote Procedure Call,” PhD Thesis, Carnegie Mellon University, Pitsburgh, Pensilvania, 1981.

[oppenhe83] Oppenheim, A., Willsky, A., and Young, I., *Signals and Systems*. Prentice-Hall, 1983.

[pickett79] Pickett, M., *ILIAD reference manual*, Computer Science Department, General Motors Research Laboratories, Warren, MI, April, 1979.

[posix90] “Information technology - Portable Operating System Interface (POSIX),” Tech. Rep., 1990.

[postel81] Postel, J., “Internet Protocol - DARPA Internet Program Protocol Specification,” Tech. Rep., RFC 791, September 1981.

[RFC10057] *RPC: Remote Procedure Call, Protocol Specification, Version 2 (RFC 10057)*, Sun Microsystems, Network Information Center, SRI International, June, 1988, Internet Network Working Group Requests for Comments, No. 10057.

[RFC1190] Topolcic, C. “Internet Stream Protocol”, Requests for Comments (RFC) 1190, CIP Working Group, October, 1990.

[reppy91] Reppy, J.H., “CML: A higher order concurrent language,” in *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation,* June 1991, pp. 293 - 305.

[rose91] Rose, M.T., *The Simple Book*. Prentice Hall, 1991.

[shapiro86] Schapiro, E., “Concurrent Prolog: a progress report,” *IEEE Computer*, 1986.

[soares92] Soares, P. G., “On Remote Procedure Call,” in *Second CASCON International Conference*, Toronto, Canada, November 1992.

[sorenson74] Sorenson, P., “A Methodology for Real-Time System Development,” Ph.D. thesis, Department of Computer Science, Univeristy of Toronto, 1974.

[stallings93] Stallings, W., *SNMP, SNMPv2, and CMIP*. Addison Wesley, 1993.

[stallings94] Stallings, W., *Data and Computer Networks*., 4th ed. New York: MAcmillan, 1994.

[stevens90] Stevens, W. R., *UNIX Network Programming*. Prentice Hall, ENglewood Cliffs, New Jersey, 1990.

[stoyenko87] Stoyenko, A., “A real-time-language with a scheduability analyzer,” Ph.D. thesis, Also available as computer systems research institute, CSRI-206, Department of Computer Science, Univeristy of Toronto, 1987.

[strom91] Strom, R.E., co , D.F.B., thur P. Goldberg, A., Lowry, A., Yellin, D.M., and Yemini, S.A., *Hermes — A Language for Distributed Computing*. Prentice Hall, 1991.

[ullman73] Ullman, J., “Polynomial complete scheduling problems,” in *Fourth Symposium on OS Principles,* 1973, pp. 96 - 101.

[yemini89] Yemini, S.A., Goldszmidt, G.S., Stoyenko, A.D., and Wei, Y.H., "Concert: A High Level Language Approach to Heterogeneous Distributed Systems," in *9th International Conference on Distributed Computing Systems,* Newport Beach, CA, June 1989, pp. 162-171.

[yemini93] Yemini, Y., Florissi, D., "Isochronets: a High-Speed Network Switching Architecture," in *IEEE INFOCOM,* 1993, San Francisco, March 1993.