TAPAS
IST-2001-34069
Trusted and QoS-Aware Provision of Application Services

TAPAS
D8: QoS Enabled Group Communication

Report version: Deliverable D8
Report Delivery Date: March 2004 (revised May 2004)
Classification: Public Circulation
Contract Start Date: 1 April 2002 **Duration:** 36m
Project Co-ordinator: Newcastle University
Partners: Adesso, Dortmund - Germany; University College London - UK; University of Bologna - Italy; University of Cambridge - UK

Project funded by the European Community under the Information Society Technology Programme (1998-2002)

D8 - QoS-adaptive Group Communication

Antonio Di Ferdinando
Paul Ezhilchelvan
Isi Mitrani
Graham Morgan

School of Computing Science
University of Newcastle upon Tyne

Contents

Summary

Practical networked systems are under increasing obligations to provide certain levels of Quality of Service (QoS) to end users. In TAPAS, we focus on the development of QoS enabled multi-party communication to support applications that require information dissemination to many processes (e.g., in auctions, distributed games). For such applications we will concentrate on QoS properties of fault tolerance, availability, and timeliness. Currently, strict separation exists between the middleware that executes application services and the network, so providing services that go beyond the best effort is difficult. Approaches to distributed system designs have thus far assumed two broad classes of computational and communication models: in the synchronous model, processing and communication delays are considered to be uniformly distributed in a known range; in the asynchronous model on the other hand, delays are finite but without any assumption on the ability to deduce delay bounds or delay distribution. So, any bound on delays, deduced however judiciously, is subject to being violated.

We here introduce a generic system model called the probabilistic asynchronous model which we claim characterises the context in which many Internet-based applications are built. Specifically, our model regards that basic services and system components (e.g., network services) do meet their performance requirements most of the time, and occasionally they may not; only when they dont, they adhere to the classical asynchronous model. Our design approach will draw from, and combine probabilistic design techniques and asynchronous ones. Its objective is to render systems that adaptively meet QoS obligations to the end users when system components meet their QoS guarantees or violate them only marginally; eventual correctness is never compromised when components fail in their QoS obligations.

Design and implantation of a QoS enabled reliable broadcast service using the principles of the probabilistic asynchronous model is described. It is possible that the QoS guarantees agreed by underlying network system are violated for a prolonged period. These violations can lead to the broadcast service being unable to meet its QoS obligations. An important aspect of a QoS enabled service is that it needs to be adaptive: it monitors the QoS offered by the underlying layer and then adapts the protocol behaviour in an effort to meet the agreed QoS to users. The QoS monitoring subsystem is thus an important part of the service. The QoS monitoring subsystem of our reliable broadcast service has been designed using the principles described in the deliverable report D10, on QoS Monitoring.

1 Introduction

The Internet is increasingly being used by organizations for offering and procuring services. Business outsourcing and application service provisioning [1] are some obvious manifestations of this trend. Obviously, when services are being traded, the networked systems providers that host such services come under obligations to offer varied levels of Quality of Service (QoS) to end users and to maintain the QoS level chosen by the users. These obligations arise from the need to retain existing customers and attract new ones in a highly competitive business environment. It therefore becomes essential that the underlying system be able to evaluate the feasibility of QoS provisioning prior to accepting an end-users QoS request and adapt to unforeseen changes in resource availability; i.e., it needs to be built *QoS adaptive*.

There are many QoS attributes that can be generally associated with a system; *latency* (or *timeliness*) and *reliability* are common ones and will be the focus of this report. Intuitively, the end-to-end QoS (e.g., latency) offered at the system level, and seen by the end user, is an aggregation (of some sort) over the QoS offered by the various subsystems that make up the system. A subsystem provides certain services to other (consumer) subsystems, by making use of the services provided to it by some other (producer) subsystems. (End-users constitute the ultimate consumer not regarded to be a part of the system.) When a service request with some specified QoS is made, a subsystem, if QoS adaptive, must, where possible, adapt its operations to accommodate the QoS request by itself; if self-adaptation alone is not possible, it should evaluate the enhanced QoS support which one or more producer subsystems need to sustain for the request to be satisfied. The request cannot be met if a producer subsystem cannot support the enhanced QoS. At the bottom-end of this producer-consumer chain are the ultimate producer subsystems that directly manage the resources themselves: communication subsystems (CS), operating systems (OS) and storage systems (SS).

Thus, building a QoS adaptive system requires that the resourceful subsystems - CS, OS and SS - offer services with QoS guarantees and dynamically respond to higher level requests for enhanced QoS support. Of these three resourceful subsystems, the service providers normally own computational resources, operating systems and storage systems. Furthermore, techniques are available for evaluating available latency, throughput and possible failure rates of OS (real-time OS in particular) and SS for a given load and operational environments. However, the situation is different with the CS if it operates on a best-effort basis over the Internet. In such an environment, the application related message traffic needs to compete for bandwidth and survive router congestions, and the CS cannot therefore offer meaningful QoS guarantees. This means that there must be means to reserve bandwidth and accord priority to traffic flows so that the CS can also offer QoS guarantees and be responsive to QoS requests.

Developments in network service provisioning indicate that such a CS can be obtained by procuring it as a service from the network service providers. Internet Service Providers (ISPs) offer to their customers QoS guarantees on the end-to-end network performance by careful network design (provisioning) based on elegant resource management models for the Internet (see [2] for example). These models take into account extensive measurements made in the past and the network providers understanding of the typical source behaviours and the

typical traffic patterns. For example, the AT&T managed Internet service a leading ISP offers 99.99% network availability, a *monthly average* latency of 60 milliseconds (within the US), and a packet loss rate of less than 0.7%.
We will assume in this report that the CS, for the purposes of building a QoS adaptive system, is provided by an ISP together with well-defined and dynamically negotiable network performance guarantees encapsulated in what the industry calls the *Service Level Agreements* (SLAs). Hosting distributed application is known to be considerably simplified by the availability of a group communication (GC) middleware system. A GC system offers many services such as reliable multicast [3, 4], consensus or atomic broadcast, and non-blocking atomic commit [5]. Many GC system have been built in the past [6]. Some assume the classical synchronous model, and most others the asynchronous model or a variation of it. Observe however that the QoS guarantees offered by the ISPs (even to the high-end users) are not deterministic, as regarded within the synchronous model; rather they are *probabilistic* in nature, admitting that the QoS metrics promised may not be met on odd occasions and that such violations would be within the remit of the stated probabilities. Note also that when the network is not guaranteed to be 100% loss-less and 100% available, a message may have to be retransmitted and may therefore take an arbitrary amount of time.
Next section will present a *probabilistic model* that characterises the environment we have considered for building a QoS GC middleware system, the *system architecture* as per which the subsystems get structured and interact, and the *subsystem architecture* identifying the internal structure of a subsystem and *the requirements* which *the protocols* and *algorithms* of a subsystem need to meet. The system architecture will conform to the principles of design composability [7]: an efficient and modular construction of a QoS adaptive system requires that its subsystems are built to be QoS adaptive as well. The subsystem architecture will lead to identification of the following requirements: the probabilistic protocols of a subsystem must be designed with configurable parameters; and, they should be associated with algorithms using which the protocol parameters can be appropriately set in order that the subsystem can be made QoS adaptive. The last two subsections of the next section are devoted to describe a reliable multicast protocol satisfying such requirements. Section 4 describes then APIs for the reliable multicast protocol just described, and the document terminates after section 5, which validates our ideas providing proof of performance and reliability of the protocol implemented.

2 The Model and System Architecture

2.1 The Probabilistic Model

The system consists of a group G of n, $n > 1$, distributed nodes that communicate using an ISP supported communication subsystem (CS). Each node host a distinct process p_i, $0 \leq i \leq n-1$. The processes of G are engaged in a multiparty coordination. They know each other's identifiers and IP addresses. Without loss of genarality, the numbering of processes is assumed to imply a 'seniority' ordering: process p_i is said to be 'more senior' than process p_j if $i < j$. A node or the process hosted within it functions correctly until and unless it crashes. A node (or a process) that does not crash is said to be correct. To present the probabilistic model, we will assume a global clock which is not accessible to processes.

- *Processing Delays*: Within a correct node, any task that is scheduled to be executed at time t, will be executed at $t + \pi$ where π is a random variable with some known distribution.
- *Storage Delays*: When a correct process initiates a storage request (for storing or retrieving of data) at time t, the request will be correctly processed at $t+\lambda$, where λ is a random variable with some known distribution.
- *Transmission Delays*: If a correct process i sends a message m to another correct process j at time t, then
  - m is delivered to j (i.e., m arrives at the buffer of j) with some probability $1-q$ (m may be lost in transmission with probability q).
  - if m is not lost, it is delivered at $t + \delta$ where δ is a random variable with some known distribution.

If the distributions of π, λ, and δ are uniform with some known mean and $q = 0$, then the probabilistic model refers to the well-known synchronous model which permits upper bounds on π, λ, and δ to be determined with certainty. The asynchronous model considers the bounds on the delays π, λ, and δ to be finite; neither the bounds nor the delay distributions can be known with certainty. For example, any bound on delays, however judiciously deduced, is vulnerable to being violated with unknown probabilities. The probabilistic model, on the other hand, assigns probabilities or coverage to quantification of delay bounds. Appendix A compares the features of the model with those of several known models.

2.2 System Architecture

The GC system lies on top of the ISP's network and the kernel is below the applications to be hosted. As per the the ISO OSI hierarchy, it can be seen at level 5 (*session*) with network (layer 4) providing a basic (unicast) communication support. It can be seen in Figure 1.

As stated earlier, GC middleware system offers a variety of services that ease application hosting. The services typically offered are:

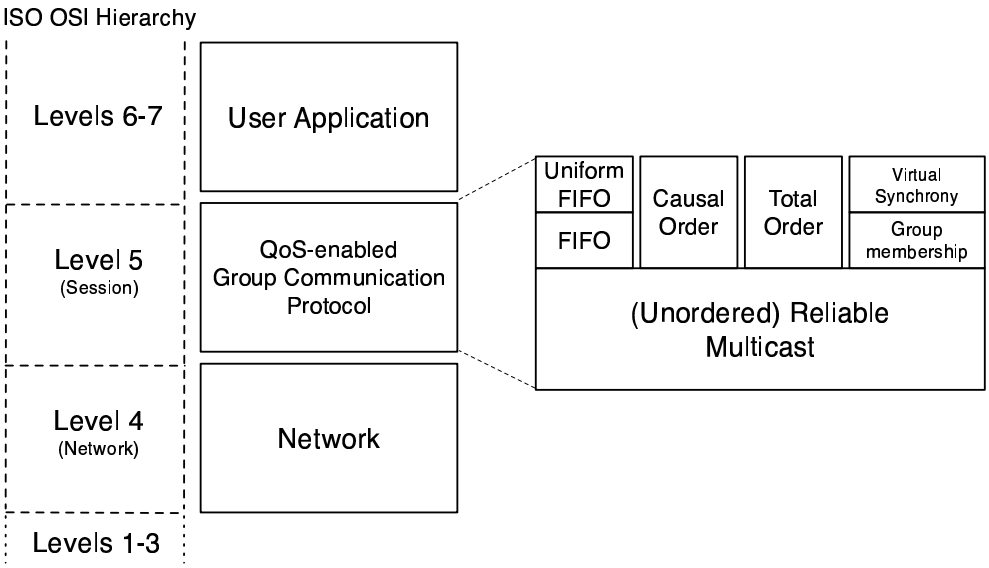

Figure 1: General architecture.

- Reliable Multicast (unordered)
- FIFO Reliable Multicast
- Uniform Reliable Multicast (unordered or FIFO)
- Causal Ordered Multicast
- Total Ordered (or Atomic) Multicast
- Group Membership
- Consensus or agreement
- Virtually Synchronous Communication [3, 12]

The (unordered) reliable multicast service ensures that a multicast m is received by either all or none of the correct members of group G and a correct members multicast is received by all other correct members. A FIFO reliable multicast ensures that a multicast m is delivered reliably and also as per the order in which the broadcaster of m sent. More precisely if m_1 and m_2 are multicast by a given process in that order, then delivery of m_2 will be after that of m_1. Similarly, the causal and total ordered services facilitate ordered delivery of some nature (see [4]) together with the multicast reliability.
Group membership service provides a realistic view to the application as to which processes are deemed to have crashed. For this view to be consistent, it needs to reach agreement (using the consensus service) with processes regarded to be operative. Virtually synchronous communication service will synchronise

the update of membership views with the messages delivered to application so that the view changes are seen by all application processes identically with respect to the set of messages delivered.
From the discussions above, it becomes obvious that a middleware service can be implemented as a service on its own (basic service) or by using other services. For example, referring to (Figure 1), the uniform FIFO reliable broadcast service can be built using FIFO reliable multicast which in turn is built using unordered reliable broadcast. In presenting the architecture for a QoS adaptive GC system, we will take a view that a given middleware service is offered using N, $N \geq 1$, services in a hierarchical manner. Obviously, the more sophisticated is the service requested by an application, the larger will be N; the unordered reliable multicast service, being the most basic middleware service, will have $N = 1$.

QoS Adaptive Middleware Architecture The model characterises the behaviour of subsystems CS, OS and SS which manage respectively the capacity to communicate, process and store information. These subsystems are collectively denoted as S_0 in Figure 2. For a QoS adaptive system to be feasible, S_0 must export a QoS management interface in addition to the traditional service interface. Using this interface, a higher-level subsystem S_1 can request S_0 whether a specified distribution for each of the delay variables (π, λ, δ) and a specified loss probability (q) can be supported; this in turn would help determine whether a given set of requirements on processing, storage and bandwidth capacities additionally needed to support an end user requirement can be met. If the request for a specified distribution for each of the delay variables cannot be supported, S_0 may respond with the delay distributions which it can currently support.
Each middleware subsystem S_i, $i \geq 1$, will have two components: a service component ($service_i$) and QoS management component (qos_i):

- $service_i$ implements a specified service tolerating at most φ node crashes;
- qos_i evaluates the delay distributions of $service_i$ as a function of such distributions offered by $service_{i-1}$. qos_i will also take into account the overhead that $service_i$ would incur given the size of input from the higher level.

At the top of the stack are the application (A) and its QoS manager (qos_A). When a user submits a request with the required (probabilistic) delay and throughput guarantees (interaction (i) in Figure 2), the application QoS manager qos_A computes and passes down the QoS guarantees expected of S_N to qos_N. The qos_N computes the guarantees expected of S_{N-1} so that the guarantees required by qos_A can be met. The guarantees expected of S_{N-1} are passed down to qos_{N-1}. The QoS feasibility evaluation thus travels down to S_0 which computes if it can maintain the necessary mean and the variance of delay distributions for the overall resource requirement. If it is possible, then the user request will be accepted; else, S_0 returns the mean and variance it can sustain and the reverse computations are made by successive qos_i upwards (interaction (ii) in the figure). The user is then informed of the QoS guarantees the system can offer.

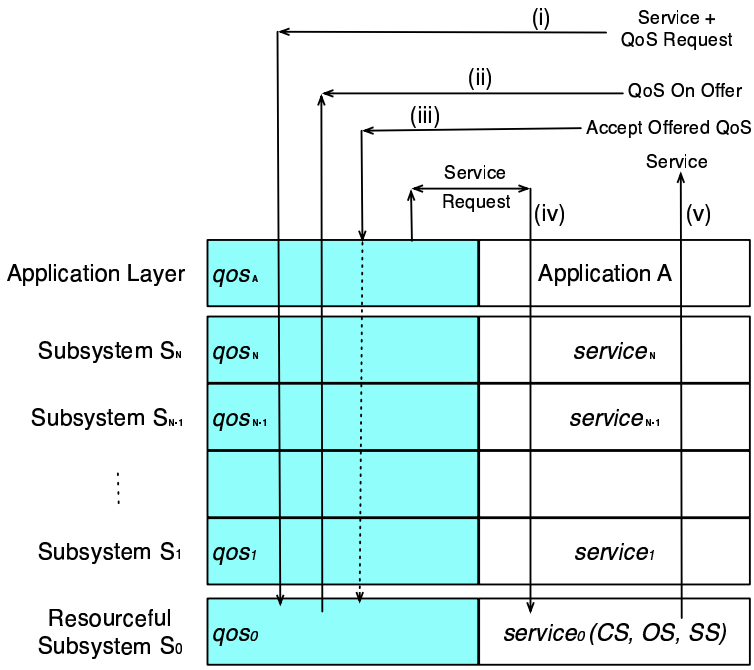

Figure 2: Structure of a Fault-tolerant QoS Adaptive System.

Structure of a Subsystem Suppose that a user request is accepted for a set of given QoS metrics. Each qos_i, $1 \leq i \leq N$, records the QoS requirements that $service_i$ needs to meet (interaction (iii) in Figure 2). The service request is submitted to the application whose execution invokes various $service_i$ (interaction (iv) in Figure 2). The (fault-tolerant) protocol that implements $service_i$ is designed with configurable parameters; the choice of these parameter values will influence the protocol behaviour and thus the QoS offered by $service_i$. These parameters can be regarded as *QoS control knobs*, and, in what follows, we will term $service_i$ as *(QoS) controllable protocol based* $service_i$, or simply as CPS_i. The responsibility for setting appropriate parameters is upon the QoS management component, qos_i, so that CPS_i (or $service_i$ in Figure 2) can meet its QoS obligations in providing its services to $service_{i+1}$. This parameter setting and the feasibility analyses carried out prior to accepting the user request will require that qos_i be equipped with algorithms to evaluate the performance of CPS_i in terms of these parameters. Specifically, qos_i should be able to evaluate the QoS metrics offered by CPS_i for a given set of parameter values (e.g., latency for a given level of redundancy) and vice versa, and also derive the parameter values from the QoS guarantees from $service_{i-1}$ below (e.g., the level of redundancy for a given loss probability) and vice versa. The module which contains these evaluation algorithms is called the (QoS) *Negotiation* module and offers a set of services called the (QoS) *Negotiation Services*. As a side remark, developing algorithms for (QoS) Negotiation module will involve stochastic modelling and performance evaluation. Tractable performance analyses generally warrant approximations to be taken and we would propose that such approximations tend to underestimate the actual performance. This means that $service_i$ will tend to perform better than predicted by Negotiation module, offering a better QoS to $service_{i+1}$ than promised. The overall system will thus have an inherent tendency not to fail on the end-to-end QoS promised to the application. It is possible that the QoS guarantees agreed by the ISP are violated for a prolonged period. These violations can lead to various higher level subsystems being unable to meet their QoS obligations at run time. So, a requirement for every given qos_i is to monitor the QoS offered by $service_{i-1}$ to CPS_i, and attempt to re-adapt the protocols of CPS_i so that CPS_i still maintains its QoS guarantees to $service_{i+1}$. The monitoring and reporting activities are carried out by the QoS Monitoring module within qos_i, and its services are collectively called the Monitoring Service. Figure 3 presents the internal structure of a middleware subsystem.

Referring to Figure 3, the interactions between the three modules of a subsystem can be summarised as below.

1. The $service_i$ is issued a service request with some specified QoS metrics shown in the figure as (1). We will suppose that the service request has already been QoS evaluated and agreed by the subsystem (see interaction (iii) in Figure 2).

2. The $Negotiation_i$ sets the appropriate parameters which the CPS_i should use to process messages related to this request (2).

3. Messages related to this request are processed by CPS_i and passed down to $service_{i-1}$ via $Monitoring_i$ (3a, 3b) which tags the appropriate downstream messages so that the tagged ones can be monitored by $Monitoring_{i-1}$.

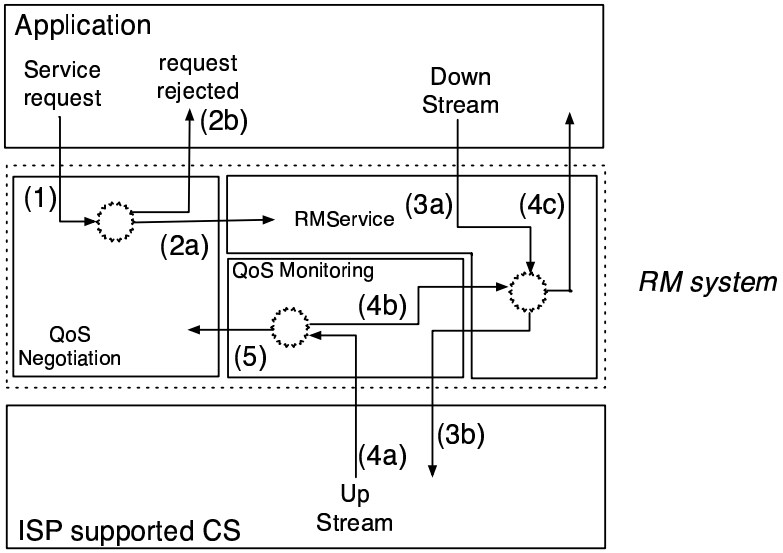

Figure 3: Layer interaction

4. Similarly, CPS_i receives the up-stream messages via $Monitoring_i$ (4a. 4b) which collects data to compute the QoS metrics offered to CPS_i.

5. The QoS metrics computed by $Monitoring_i$ are periodically sent to $Negotiation_i$ (5) which in turn re-adapts the parameter values of CPS_i if necessary and if possible.

2.3 Layer Structure

Summarizing what said so far, the protocol has a layered architecture, arranged in such a way to form a stack. Each layer offers a different service, that relies on the service offered by the lower layer. The bottom-most layer offers an unordered reliable multicast service, while the top-most layer offers more sophisticated services such as total or FIFO ordering service. The full, most complete service is obtained by going through the whole stack.
A single layer is composed by three sub-services: the Core Protocol Sub-service realizes the service offered by this layer by processing both incoming and outgoing data. It relies, for its correct execution, on parameters generated after successful negotiation of QoS service with the user by the Negotiation Subservice. Such parameters represent a guarantee that agreed QoS service can be achieved, validated by a constant monitoring of the QoS Monitoring Subservice whose aim is to probabilistically model a certain set of resources that will provide the basis for evaluations in the Negotiation Sub-service.

Figure 4 shows an example of a protocol providing FIFO Reliable multicast service. Two observations can be made: the first is the position of the QoS Monitoring Sub-service inside a layer. At reliable multicast layer it is really

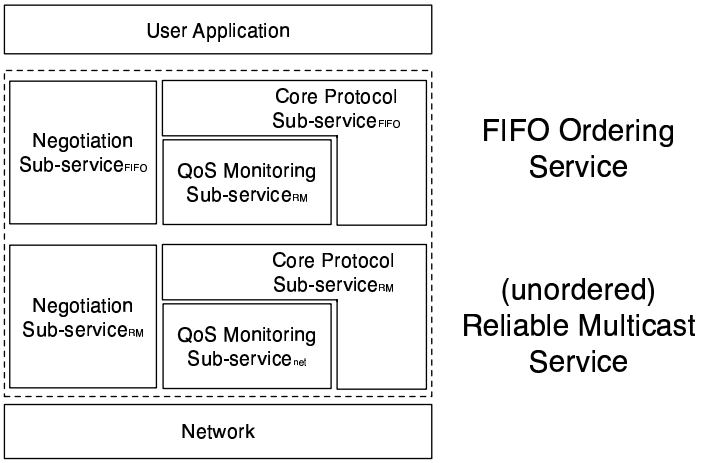

Figure 4: Protocol for Total Ordering

monitoring the lower layer, Network in the picture. Reliable multicast activity is then monitored at FIFO layer. The idea behind this mechanism of monitoring is that a layer relies, for its own execution to be correct, on the lower layer, and is so in its interest to directly monitor that no mistakes or faults occur at that layer.
The second observation is that QoS Monitoring Sub-service at reliable multicast layer is an extra security adopted by the protocol itself. In an ASP context, in fact, the layer called Network in the picture would be substituted by an ISP, which would typically provide all needed data about the network. We could, then, choose to trust in data provided by the ISP rather than using the QoS Monitoring Sub-Service at reliable multicast layer. This is an important consideration because means that should the QoS Monitoring Sub-service fail, we could still guarantee adaptation inside the reliable multicast protocol.

2.4 Architecture for Mutual Monitoring

Figure 5 shows an example of mutual monitoring. The *Core Protocol Sub-Service* (CPS) is responsible for implementing the logic of the Group Copmmunication System (GCS) and providing user access to reliable multicast. We have described the user in our example as *accessing services* (AS). Such services may be considered the application or protocols responsible for higher level message guarantees (e.g., message ordering). The CPS requires access to underlying network services to enable message dissemination across computer networks. We describe such services as *network services* (NS). An NS may provide QoS guarantees to the CPS (e.g., mean message delivery latency) and so may directly influence the way the CPS functions.

The monitoring requirement is satisfied by *Metric Collectors* (MeCo). A MeCo is co-located with a protocol layer (identified by subscript) and is re-

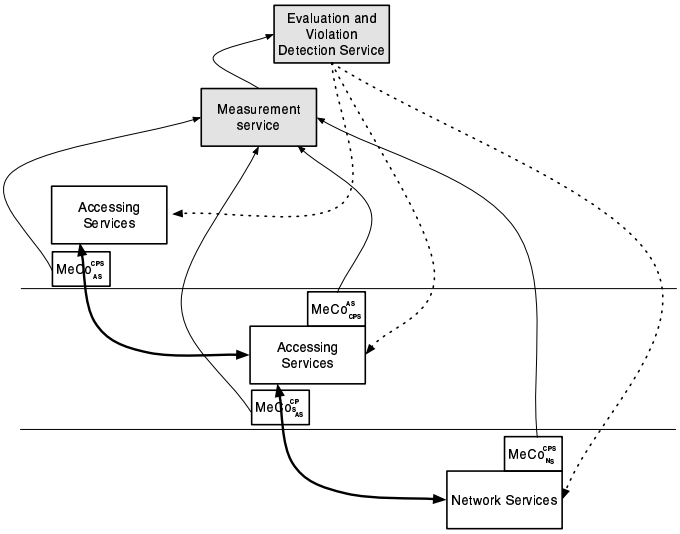

Figure 5: Deployment of Mutual Monitoring of QoS for reliable Multicast.

sponsible for monitoring the QoS of a protocol layer (identified by superscript). A MeCo collects QoS metrics and passes them to the *Measurement Service*. The measurement service then correlates information gained by one or more MeCos and provides a suitably formatted message for consumption by the *Evaluation and Violation Detection Service*. The evaluation and violation detection service is responsible for informing protocol layers of SLA violations.

3 Design and Description of a Reliable Multicast Protocol

This section presents the probabilistic Reliable multicast protocol, the most basic protocol in our QoS adaptive GC system. The protocol is designed with configurable parameters using which QoS guarantees on offer can be set to the desired level. The guarantees are obviously probabilistic in nature and fall into two broad catagories: *reliability* and *latency*. We first recap the system context and then present the assumptions which the protocol design makes.

1. In a group G, of distributed processes numbered $1, 2, \ldots, n$: $G = \{p_1, p_2, \ldots, p_n\}$, for some known $(n > 1)$, a process can either be opeartive or crash to be inoperative permanently. A process that does not crash during an execution of the protocol is said to be *correct* in that execution.
2. Processes know each other's numbers and the numbering of processes implies a 'seniority' ordering: process p_i is said to be 'more senior' than process p_j if $i < j$.
3. Each process has a primitive $send(m)$ using which it can send a message m to another process.
4. The $send(m)$ is successful if m is deposited in the receive buffers of the destination process.
5. The communication subsystem (managed by an ISP) assures that (i) a $send(m)$ operation is successful with a known probability $1 - q$, i.e., m is lost with probability q; and (ii) the transmission delay of a successful $send(m)$ operation is an independent random variable with some known distribution.

We make the following simplifying assumptions:

- Processes are over-provisioned on computational and communication capacity. Consequently, a process can instantaneously receive a message which the communication subsystem deposits into its receive buffer. This means that the message transmission delay will be the inter-process communication delay.
- For the purposes of simulations, the transmission delay distribution will be assumed to be exponential with mean d.

3.1 Specification of Protocol Guarantees

The protocol exports two primitives: $RMCast(m)$ and $RMDeliver(m)$. When a process wishes to multicast a message reliably to processes in G, it invokes the operation $RMCast(m)$. This process will be called the *originator* of m. A message m sent by invocation is delivered to a destination process by $RMDeliver(m)$. The protocol offers the following reliability guarantees (in probabilistic terms):

1. $Validity$ If the *originator* of m does not crash until its invocation of $RMCast(m)$ is complete, then all operative destination processes deliver m with a probability V which can be made arbitrarily close to 1 (by appropriate choice of parameter values).

2. *Agreement* (or $Unanimity$). Irrespecive of whether or not the *originator* of m remains operative to complete its invocation of $RMCast(m)$, if a destination process delivers m then all correct destination processes deliver m with a probability A which can be made arbitrarily close to 1.

We note here that the *agreement* guarantee actually refers to the $uniform$ *agreement* property: even if a destination process crashes shortly after delivering m, then all correct destinations are guaranteed to deliver m. This means that if the crashed process has invoked $RMCast(m')$ soon after delivering m, then any correct process that delivers m' is guaranteed to deliver m as well. Observe that A will be 1 if $n = 2$, i.e., if there is only one destination process. The protocol offers the following guarantees on latency metrics.

1. The interval between an originator invoking $RMCast(m)$ and the first instant thereafter when all correct destination processes have received m, does not exceed a given bound, D, with a probability, r_D, which can be evaluated in advance.

2. If, following an invocation of $RMCast(m)$, the message arrives at a correct process, then it will arrive at all other correct processes within a further interval of a given length, S,with a probability, u_S, which can be evaluated in advance.

These properties are sometimes referred to as *latency bound* and *relative latency bound*, respectively. They enable an application developer to reason about timeliness: an application process that invoked $RMCast(m)$ at time t, can be programmed to regard at time $t + D$ that m is delivered to all correct destinations; a destination process that has delivered m (through $RMDeliver(m)$) at time t can be programmed to regard at time $t + S$ that all correct destinations have delivered m.

3.2 QoS Feasibility Evalutaion and Adaptation.

From the description above, it can be seen that the probabilistic guarantees offered by the protocol can be evaluated in advance given a set of parameter values or can be changed to the desired effect. This aspect is utilised by the associated $Negotiation$ module to evaluate the feasibility of QoS support required by a request. For example, an application that wishes to perform a reliable multicast can specify the desired success probability, R, and the latency bound, D. The interface would respond by evaluating r_D and comparing it with R: if $r_D \geq R$, then the specification is achievable; otherwise not. Clearly, the larger the value of D, the higher the achievable probability of success. Similarly, a user or an application that wishes to be delivered a reliable multicast can specify a desired success probability, U, and a relative latency bound, S. The interface would evaluate u_S and compare it with U: if $u_S \geq U$, then the specification is achievable; otherwise not.

Moreover, two forms of adaptation are possible at run time:

Adaptation for reduced message overhead. The evaluation of r_D and u_S involves taking approximations for resaons of analytical tractability. These approximations are deliberately chosen to have a bias for underestimate the evaluated performance. During the course of a protocol execution, a process is

equipped to sense that the protocol is performing better than expected for a given R, D or U, S, and adapt parameters that would result in smaller message overhead.

Adaptation to observed QoS perturbs. When QoS monitor reports that the communication subsystem is not maintaining the promised QoS metrics, then the protocol may not be able to sustain R, D or U, S which seemed plausible during the QoS feasibility evaluation. The protocol offers parameters which, when reset appropriately, can avoid failing to meet a given R, D or U, S due to unexpected QoS perturbs.

These claims on QoS feasibility evaluation and parameter adaptation will be examined through simulations.

3.3 Protocol description

The reliable multicast protocol has three features which are designed to assure high probability of success at tolerable cost in message traffic:

- (a) The execution of $RMCast(m)$ comprises more than one invocation of a $broadcast(m)$ operation. Each of these invocations concurrently sends the message m once to each destination.
- (b) The responsibility for invoking $broadcast(m)$ initially rests with the originator of the message, but may devolve to another process, and then to another, in consequence of crashes, message losses or excessive delays.
- (c) In the event of such a devolution, a decision procedure attempts to select exactly one process to take over the broadcasting responsibilities.

These features can be described as *Redundancy*, *Responsiveness* and *Selection*, respectively. The *Redundancy* of the protocol is controlled by two parameters:

- An integer, ρ, specifies the level of redundancy; the originator of a message makes $\rho + 1$ attempts to broadcast it (if operative); these attempts are numbered $0, 1, \ldots, \rho$; typically, $\rho \geq 1$.
- The interval between consecutive broadcasts is of fixed length, η; that length is chosen to be as small as possible, but sufficiently large to make any dependencies between consecutive broadcasts negligible.

One way of choosing η is to require that the transmission delay between a source and a destination is less than η with a given probability, α (reasonably close to 1). In the case of exponentially distributed delays with mean d, η is given by

$$\eta = -d \log(1 - \alpha) \; .$$

More conservatively, η can be chosen so that it exceeds the *largest* of $n - 1$ parallel transmission delays with probability α. In the exponential case, that choice would imply

$$\eta = -d \log(1 - \alpha^{\frac{1}{n-1}}) \; .$$

Responsiveness. If the originator of a message crashes during its redundant broadcast attempts, the destination processes respond by taking over the broadcasting responsibility upon themselves. To facilitate this takeover, each

copy of a message, m, has fields $m.copy$, $m.originator$ and $m.broadcaster$; these specify the number of the current broadcast attempt $(0, 1, \ldots, \rho)$, the index of the originating process, and the index of the process that actually broadcast the message m, respectively. The values of $m.originator$ and $m.broadcaster$ will be different if a destination process carries out the broadcasting of m.

Every process that receives a message, m, such that $m.copy = k < \rho$, must be prepared to become a broadcaster of m if necessary. It does so by setting a timeout interval of length $\eta + \omega$, with some suitable value of ω (η is the interval between consecutive broadcasts, while ω accounts for differences in transmission delays, or 'jitter'). If copy $k + 1$ of m arrives from the broadcaster of copy k before the timeout expires, then all is well with that broadcaster; the receiver process sets a new timeout of $\eta+\omega$ for the next copy (if there is one). Otherwise, the receiver pessimistically assumes that the process $m.broadcaster$ has crashed while broadcasting copy k of m, and that it is the only process to have received any copy of m. It therefore prepares to appoint itself as a broadcaster of copies $k, k + 1, \ldots, \rho$.

However, the $m.broadcaster$ may not in fact have crashed; copy $k + 1$ of m may just be delayed unduly or lost; moreover, even if $m.broadcaster$ has crashed, this receiver may not be the only process that has observed the crash. In order to avoid multiple receivers becoming broadcasters unnecessarily, a further random wait, ζ, uniformly distributed on $(0, \eta)$, is added to the timeout interval $\eta + \omega$. If a copy number k or higher is not received before the expiration of ζ, this receiver appoints itself as a broadcaster. Otherwise it sets a new timeout of $\eta + \omega$.

Selection. The protocol guards against multiple self-appointed broadcasters. It requires that any broadcaster with index i, whose latest broadcast has been of copy k of the message, should relinquish its broadcasting role in any of the following circumstances:

- 1. Process i receives a message m such that $m.copy = k$ and either $m.broadcaster < i$ or $m.broadcaster = m.originator$. That is, a more senior process has assumed the duties of broadcaster, or the originator has not in fact crashed.
- 2. Process i receives a message m such that $m.copy > k$. This would happen if process i has missed one or more copies of m, and now learns that another broadcaster is closer to completing the protocol.

Suppose that process i has abandoned its broadcasting role and has set a timeout expecting a copy, say, k, from broadcaster j. It will have to reset that timeout if either copy k is received later from a broadcaster more senior than j or from the originator, or copy $k + 1$ or higher is received from any broadcaster. This is necessary because when process j receives the message which process i has just received, it would relinquish its broadcasting role.

The purpose of these provisions is to avoid unnecessary broadcasts and hence message traffic, while still making the best effort to ensure that $\rho + 1$ copies of each message are broadcast. The idea is that when any broadcaster crashes, all receivers that time out on $\eta + \omega + \zeta$ will briefly become broadcasters, but after that only one of them is likely to continue broadcasting, at intervals of length η. That process will be a receiver process if the originator has crashed or its messages suffer excessive delays. A more detailed pseudo-code description

of the reliable multicast protocol executed by the process i is presented in the following subsection and in figure 1.

3.4 Details of the protocol

An execution of $RMCast(m)$ starts by setting the field $m.originator$, and also a unique message identifier called $m.sequenceNo$; then $(\rho + 1)$ invocations of $broadcast(m)$ are performed, with $m.copy = 0, 1, \ldots, \rho + 1$. The primitive $broadcast(m)$ sets the $m.broadcaster$ field and concurrently sends m to all other processes in G.

The protocol for delivering a reliable multicast message is $RMDeliver()$, and is structured into two concurrently executed parts. The first part handles a received message and the second part the expiry of timeout $(\eta + \omega)$. Three integer variables are maintained for a received message m distinguished by $m.originator$, $m.sequenceNo$:

- $max\_recd_i(m)$ has the largest copy number received for m.
- $leader_i(m)$ has the index of the process from which m with copy $max\_recd_i(m)+1$ is expected.
- $last\_own\_bcast_i(m)$ contains the copy number of m which the process i broadcast when it last acted as a self-appointed broadcaster.

A received message calls for one or more of the following three actions:

- New m. Variables are initialised and m is delivered (lines 6-12).
- $m.copy = \rho$. Blocks any future occurrence of the third action (described next), by setting $max\_recd_i(m)$ to ∞ (MaxInt) (line 13). Note that a new m can have $m.copy = \rho$ if earlier copies are lost or excessively delayed.
- *Change of* $leader_i(m)$: The received m indicates one of the circumstances (described earlier) in which the process i needs to either relinquish its broadcasting role or change the broadcaster from which the next copy is expected. A new timeout $(\eta+\omega)$ is set after $max\_recd_i(m)$ and $leader_i(m)$ are updated (lines 14-20).

When timeout $(\eta+\omega)$ for m expires, an additional timeout ζ is set, during which a message with appropriate copy number from any broadcaster is admissible. So, $leader_i(m)$ is set to MaxInt (line 21). If no such message is received, process i appoints itself as a broadcaster and sets up a thread $Broadcaster(m)$ (lines 22-24). The thread $Broadcaster(m)$ broadcasts m only if the process i remains to be the broadcaster (i.e., $leader_i(m) = i$) as per selection rule; otherwise, it dies (lines 25-32).

Observation *A process will use at most one* $Broadcaster(m)$ *thread for a given* m *at any time.* Suppose that the $timeout(m)$ expires successively at times t_1 and t_2 for process i. The thread created at t_1 must die before t_2 for the following reasons.

A thread can be created only after the timeout for $\eta + \omega$ and then another one for ζ have expired. The timeout for $\eta + \omega$ is set only when $leader_i(m) \neq i$

(line 18). A thread dies within η time after $leader_i(m) \neq i$ becomes true (lines 25, 28). Since $\eta + \omega + \zeta \geq \eta$, the first thread dies by the time the next one is created. The thread pool in practical systems is not unbounded. So, our protocol makes judicial use of the available threads.

4 API description

The programming language chosen to develop the protocol is Java. This choice is mainly driven by the Java-native nature of the JBoss application server, used as reference platform for the TAPAS project.
As said earlier in this document, each layer is composed out by three separated sub-services that work in tight coordination to offer a reliable service under respect of required (and agreed) timely guarantees. In order to reflect such a separation, each sub-service has been implemented as part of a bigger java `package`, whose structure is shown in Figure 6. Naming conventions are taken from the standard java package naming system[1]. According to these, the package has been named `uk.ac.ncl.cs.`*`subServiceName`*, where *`subServiceName`* is the name of the sub-service. The protocol makes use of the *interface programming* model, that implies an object to be defined by a Java interface generalizing object's properties and letting it be implemented by another method tied to the context the object is going to be used in. This technique permits a clear separation of the object's definition from the actual implementation, allowing adaptation of an object's definition in many contexts and then reusability. Moreover, interface programming allows a clear modularization of the architecture, letting the programmer to choose between several ways of implementing the same object.
In the context of the protocol, general objects have been firstly defined by means of Java interfaces, while implementation has been thought in the context of the layer to implement. This makes the protocol highly extensible, since allows for new services to be easily added to a stack. Each sub-service of the layer is defined by an interface[2]. These interfaces have been named `NegotiationService`, `CoreProtocolService` and `MonitoringService`, and their implementation is layer dependent. In addition to the three sub-services, a sub-package offering external services has been created, where at the moment resides a basic protocol for group formation and management, called `GroupManager`, and a violation detection service, called `ViolationDetector`.

In our implementation each protocol layer exhibits interfaces via CORBA RPC. Via such interfaces, a protocol layer may access message dissemination services of the protocol layer directly beneath them in the protocol stack. A CORBA call back mechanism is used by a protocol layer to deliver messages to protocol layers immediately above them in the protocol stack. Data relating to the metrics of QoS is passed to the measurement service via the *Java Messaging Service* (JMS) by MeCos and then by the measurement service to the evaluation

[1] The Java standard package naming conventions requires, in order to guarantee uniqueness, a non-core package to be named by the network domain of the machine, followed by the chosen name of the package.

[2] Throughout this document, we will refer to the word *interface* in the Java perspective, meaning a Java interface.

```
RMCast(m)
(1)    m.originator ← i; m.SequenceNo ← seq_number;
(2)    m.copy ← 0;
(3)    repeat(ρ + 1) times →
(4)      {broadcast(m); wait(η); m.copy ← m.copy + 1;}
```

```
RMDeliver()
    begin
     cobegin                          // message-handling part
(5)       receive(m);
(6)        if new(m) →
(7)          begin
(8)            max_recd_i(m) ← m.copy;
(9)            leader_i(m) ← m.broadcaster;
(10)            last_own_bcast_i(m) ← −1;
(11)            deliver(m);          // m is delivered (once) to the application
(12)          end

(13)       if (m.copy = ρ) → {max_recd_i(m) ← maxInt;}

(14)       if(m.copy > max_recd_i(m)) ∨
(15)         (m.copy = max_recd_i(m) ∧
           (m.broadcaster = m.originator ∨ m.broadcaster < leader_i(m)) →
(16)         begin
(17)           max_recd_i(m) ← m.copy;
(18)           leader_i(m) ← m.broadcaster;
(19)           set timeout for η + ω;
(20)         end
     coend
     cobegin
                  // timeout-triggered, timer-driven part
       timeout(m)⟶
        begin
(21)          leader_i(m) ← maxInt;
(22)          wait(ζ) ;
(23)          if leader_i(m) = maxInt →
(24)           {leader_i(m) ← i; create thread Broadcaster(m);}
        end
     coend
    end
```

```
Broadcaster(m)
    begin
(25)    while((max_recd_i(m) < ρ) ∧ (leader_i(m) = i)) do
(26)        m.copy ← max{last_own_bcast_i(m) + 1, max_recd_i(m)};
(27)        broadcast(m);
(28)        max_recd_i(m) ← m.copy;
(29)        last_own_bcast_i(m) ← m.copy;
(30)        wait(η)
(31)    od
(32)    die; // the thread dies.
    end
```

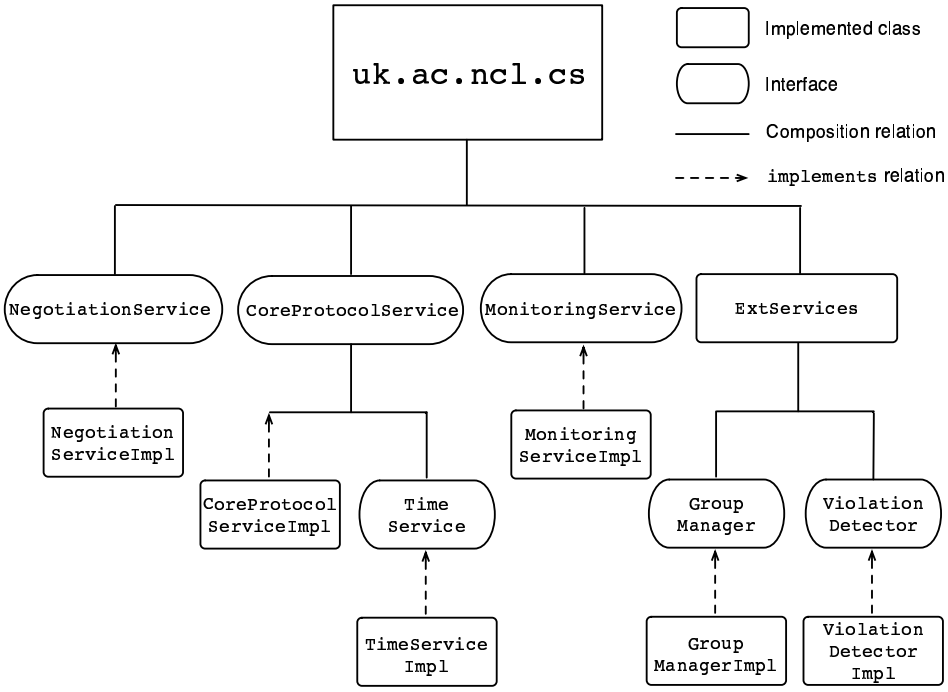

Figure 6: Package structure

```
module CoreProtocolService{
    struct groupMember{
        string memberID;
        string IPAddress;
        short portNumber;
    };

    typedef sequence<groupMember> targetDelivery;
    typedef sequence<any> deliveredMsg;

    interface RMGroup{
        void RMCast(in targetDelivery, in any msg);
        void RMDeliver(out deliveredMsg);
    };
};
```

Figure 7: CORBA IDL from CPS

and violation detection service via JMS. The evaluation and violation detection service passes notification of violations of an SLA to interested parties (protocol layers) via JMS. Figure 7 shows the CORBA IDL for the core protocol service.

Group lifecycle: Two interfaces are provided that provide access to group lifecycle and message handling services for clients. The CPS interface provides two methods:

- **RMCast** Allows clients to issue multicast messages to a particular group (used by AS).
- **RMDeliver** Allows delivery of messages to CPS (used by NS).

We chose CORBA RPC for inter-protocol layer communication to enhance the interoperability of our system and to enable a MeCo to be integrated into our service in a non-intrusive manner via CORBA interceptors. Interceptors enable the interception of messages (down calls and up calls) without any change to application logic. Via the use of interceptors a MeCo may obtain metric measurements related to QoS of a protocol layer. For example, $MeCo_{cps}^{NS}$ allows the gathering of metric data relating to the performance exhibited by the NS layer as viewed by the CPS. We chose JMS for passing messages between the measurement/evaluation services and the protocol layers as such communications are message oriented and may be consumed as and when appropriate with minimal impact on performance and so promote a scalable solution. For example, there may be many instantiations of different protocol layers requiring similar message type communications with the measurement service. Rather than require synchronous RPC on a per-protocol layer basis (a non-scalable solution) a more appropriate approach would be to enable messages to be passed to the measurement service via event channels (provided by JMS) that are associated to particular message types (we assume different instances of protocol layers

```
<?xml version="1.0" encoding="UTF-8">
<ExchangeDoc>
    <pars>
        <eta value="4.60"/>
        <rho value="2"/>
        <omega value="0"/>
    </pars>
</ExchangeDoc>
```

Figure 8: Example of performance metrics described in XML

would use the same event channels). Figure 4 provides a sample message that would be passed via JMS relating to the performance metrics. This message would be exhibited by the CPS and relates to the settings that govern the behaviour of the RMCast protocol. We now provide a detailed description of the different monitoring and evaluation present in the system. As our primary concern is the CPS, we concern ourselves with monitoring that may directly influence evaluations and violations that may impact the function of the CPS.

Monitoring & Evaluation: The overall performance of the CPS is influenced by the QoS provided by the NS and the usage made of the CPS by the AS. Therefore, the following MeCo are required for determining SLA violations in the system

- $MeCo_{AS}^{CPS}$ Co-located with AS and responsible for monitoring the QoS provided to the AS by the CPS. These metrics are based on the interception of messages between the AS and the CPS using CORBA interceptors.
- $MeCo_{CPS}^{AS}$ Co-located with CPS and responsible for monitoring the usage the AS makes of the CPS. These metrics are based on information supplied directly from the CPS.
- $MeCo_{CPS}^{NS}$ Co-located with CPS and responsible for monitoring the QoS provided to the CPS by the NS. These metrics are based on the interception of messages between the CPS and the NS using CORBA interceptors.
- $MeCo_{NS}^{CPS}$ Co-located with NS and responsible for monitoring the usage the CPS makes of the NS. These metrics are based on information supplied directly from the NS.

From our descriptions we may determine two basic types of MeCo a protocol layer may require: (i) aid in determining if a protocol layer is used inappropriately, (ii) aid in determining the QoS provided to a protocol layer by a lower protocol layer. As described above, type (ii) uses CORBA interceptors to gain the relevant metric data. Type (i) requires a protocol layer to exhibit an interface that allows a MeCo to gather information as and when required. Such an interface is based on XML message exchange. We use SOAP based messages to transfer this information from a protocol layer to an associated MeCo of type (i). Periodically a MeCo constructs appropriate summary information based on the

metric data gathered and prepares a message in the form of XML for passing, via JMS, to the measurement service. We use XML as the evaluation and violation service uses XML based language constructs for determining if SLAs have been violated. The measurement service assumes responsibility for correlating the QoS data received from a number of MeCo instances into a form appropriate for acceptance by the evaluation and violation detection service. This is required as the tailoring of such information is dependent on the instances of SLAs that govern QoS between multiple instances of clients and servers (protocol layers). For example, different instantiations of a protocol layer may exist on a per client application basis, each with their own SLA. Placing the tailoring of QoS information at the MeCo level would require an instantiation of a MeCo on a per-client basis. However, by having a per-protocol layer type MeCo gathering QoS information we can construct the appropriate information in the measurement service. This allows a protocol layer to only require a single MeCo, irrelevant of the number of clients (higher protocol layers) associated to it. This is a more scalable solution as the MeCo appears light-weight in the fact that the unnecessary processing burden related to the many different SLAs a protocol layer may be participating in is confined to the measurement service. Protocol layers register their interest to event channels (provided by JMS) on a per-SLA basis. Violation of an SLA results in the evaluation and violation detection service issuing an XML message detailing the type of violation that has occurred on the appropriate SLA event channel. The responsibility of consuming such messages is left to the individual protocol layers. This decoupled communication is ideal in that the evaluation and violation service does not have to contact directly each protocol layer that is associated to an SLA. Once protocol layers have consumed messages indicating SLA violation the negotiation process between protocol layers may be enacted. Such negotiation is protocol layer dependent and is detailed in previous literature related to CPS, AS and NS.

4.1 APIs for reliable multicast protocol

Reliable multicast is the bottom-most layer of the architecture, and the algorithm implemented offers a QoS-adaptive reliable multicast service. If you recall from Section 2 that a single layer relies, for its own execution, on the lower layer, it's easy to understand how each layer up in the architecture relies either directly or indirectly on this layer and then how important this layer is. Primitives exported by this layer are aimed to multicast messages according to the protocol's specifications, and are then `RMCast(GroupRef ref, Object msg)` for the sending part and `RMDeliver()` for the receiving part. Real network communication has been defined, as well as all other general objects, by means of interfaces, that in the current release have been realized by means of datagrams sent over UDP `DatagramSockets`. Characterization of all three sub-services at this layer, as well as APIs, are explained in detail in the rest of this section.

4.1.1 Negotiation Sub-service

Negotiation Sub-service's primary aim is to negotiate a QoS service level with the user and evaluate parameters for Core Protocol Sub-service's execution.

```
public interface Negotiation {
    public boolean negotiate(double successProb, double delay,
                             String type, GroupRef ref);
}
```

Figure 9: `Negotiation` interface

Its structure implements the `NegotiationService` interface by spanning two threads, one providing API for the real negotiation, and another to listen to models updates coming from the QoS Monitoring Sub-service. The first thread, named `Negotiator`, is needed for the real negotiation with the user, and simply calls the `Negotiation` interface to negotiate QoS based on this layer's service. In the reliable multicast layer, this interface's implementation contains all analytical approximation formulas needed to evaluate probability of success based on the user's requested delay. The `Negotiation` interface has then the form shown in Figure 4.1.1:

where `succesProb` and `delay` are user's requested probability of success and delay respectively, `type` is the type of delay requested, either `absolute` or `relative`, and `ref` is a reference to the Group Manager, needed to get any information about the group.
The second thread, named `ModelUpdater`, has the aim of taking track of information coming from the QoS Monitoring Sub-service and eventually update current probabilistic models for resources upon detection of changes. Such updating is made by means of the `UpdateModel` interface, that simply calls the layer-dependent method `update`. Figure 10 shows the Negotiation Sub-service's structure.

4.1.2 Core Protocol Sub-service

The Core Protocol Sub-service realizes the true service offered by this layer, and at bottom-most layer offers a QoS-adaptive reliable multicast algorithm. Since a multicast protocol must deal with both sending and receiving a message, two are the main APIs provided by this sub-service. At sending side, the main primitive `RMCast(GroupRef ref, Object msg)` allows the caller to reliably send a message according to the reliable multicast protocol's specifications described in Section 3, while receiving side's main primitive is `RMDeliver()`, that allows the caller to reliably deliver a message.

Figure 11 shows Core Protocol Sub-service's structure. Upon instantiation, the `CoreProtocolservice` spans a thread for the sending part of the reliable multicast protocol (`sendThread`) and a thread for the receiving part (`ReceiveThread`) that, in turn, call the `RMCast(GroupRef ref, Object msg)` and the `RMDeliver()` respectively. The former thread waits for the user to request the sending of a message, while the latter starts to listen to a port for incoming messages from other members of the group.
Both primitives for reliable sending and receiving a message are described in details in the next paragraphs.

Reliable Multicast Operation: The Reliable Multicast operation is implemented by the `RMCast(GroupRef ref, Object msg)` method. The real mes-

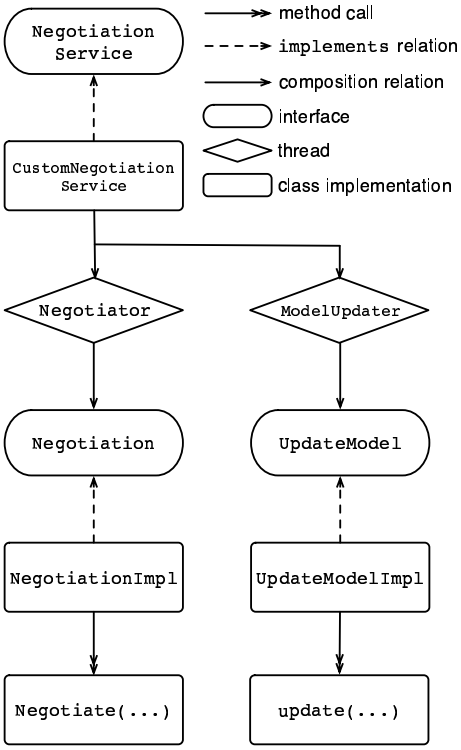

Figure 10: Negotiation Sub-service structure

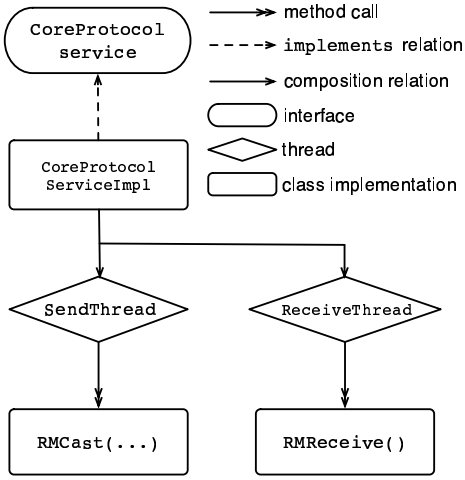

Figure 11: Core Protocol Sub-service structure

```
public class Message implements java.io.Serializable{
    String originator;   // Message's originator
    String broadcaster;  // Message's broadcaster
    int copyNo;          // Level of redundancy realized
                          // by this Message
    String msg;          // Real message to send
    long tStamp;         // Timestamp for this message
    long uniqueProcessId;// Unique ID for this multicast operation
    int sequenceNo;      // Sequence no. for this message
    int rho;             // Level of redundancy
    double eta;          // Gap time for failure independence
}
```

Figure 12: Format for the `Message` data type

sage is wrapped into a custom data type named `Message`. The form for this data type is shown in Figure 4.1.2

The `sequenceNo` field has been included in case of fragmentation of the real message. Level of redundancy realized by the message, `copyNo`, must be $0 \leq copyNo \leq \rho$, while the overall level of redundancy, `rho`, must be $\rho \geq 1$. All other fields are self explanatory.
To send a message according to the reliable multicast protocol, when called, the `RMCast` method invokes a generic `Multicaster` interface that in this context is implemented by a specific `UDPMulticaster` class. This class is basically composed out by a buffer where the message to send is deposed, and a pool of threads that send the message to all other group members. This structure is shown in Figure 13.

Threads inside the pool are given a shared synchronized list of recipients to send the message to. A thread that has to send this message, gets a copy of the message to send and extracts the head from the recipients list. It then sends the message to the member referred by the element of the recipient list just extracted, after which it throws the element away and extracts the current head from the list. This process is repeated until the recipients list is empty, after which all threads in the pool terminate. The purpose of having a pool of threads to send a message instead that a single one is that in this way we can approximate reasonably well a concurrent message sending. According to the protocol's specifications, after some time (η in the protocol description), the process is repeated to send the next copy of the message. The whole process of sending a message according to the reliable protocol specifications terminates when the original message has been sent $\rho + 1$ times.

Reliable Delivery Operation: Reliable Delivery operation's main primitive is `RDeliver()`, and the main mechanics strictly follows the pseudo code described in 3.4. All probabilistic data required for the execution (starting value of ω, for instance) is obtained from the Negotiation Sub-service. Reception of a message (Figure 14) consists in a `Receiver` receiving the message from the network. `Receiver` is of course an interface, implemented by the `DatagramReceiver` class that opens a `DatagramSocket` on some port and

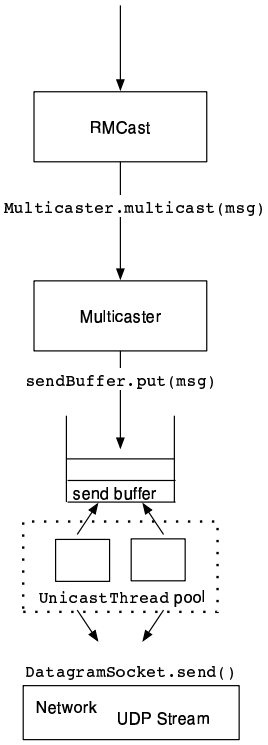

Figure 13: Reliable Multicast operation

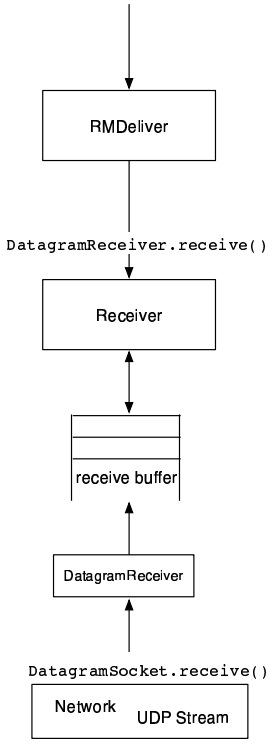

Figure 14: Reliable Delivery of a message

listens for incoming messages. `DatagramReceiver` puts then the message in a receive buffer, from which the `RMDeliver` method pops it and treat it according to the reliable protocol specifications. At this level, upon reception of a copy of a multicast operation, all support data structures are updated, and the message is stored on a so called *Message Bag* (`MsgBag`). This structure's purpose is to store received messages, that might be useful to have in determinate situations. It is realized as a linked list of `Message`s. This list is ordered on the copy a message refers to (`copyNo` field of the `Message` data type) and, in case of equality on the process broadcasting the message (`broadcaster`) field, to efficiently check whether a copy has already been received or not. Since the protocol is thought as managing more than a multicast operation at a time, `MsgBag`s are themselves stored in a *Message Bag Repository* (`MsgBagRepository`) ordered on the unique multicast ID. After storing a received message, the process has to set a timeout by which to expect the next copy of a message. This is done by means of a custom *Time Service*, described below.

Time Service: The time service is defined by the `TimeService` sub-package, which exports a `TimeService` interface implemented by the `TimeServiceImpl`

```
public class TimeoutListener extends EventListener{
    static long originalTimeout; //timeout to get notification for
    static String owner;         //owner of the listener
    static long clockedTimeout;  //timeout converted into the time
                                 //service clock format
    static long serviceTstamp;   //timestamp assigned by the owner
    static int copyNo;           // copy the timeout refers to
    static long multicastID;     // unique ID of the multicast
    static int ownerPort;        // port of the listener's owner
}
```

Figure 15: `TimeoutListener` object

class. Despite the fact that a time service is typically located on a remote machine, for our purposes the time service has been thought to be locally collocated in the same machine hosting the Core Protocol Service (and the whole complete protocol architecture). The main reason for this choice is that notification of a timeout expiration is required to be very timely, since there is a QoS requirement to respect. What we really needed, therefore, is a separate time service for each member of the group. Besides the pure timely reasons, local placement of the time service on the same host hosting the whole sub-service allows us not to introduce any failure prone link between the Core Protocol Sub-service and the time service.
In the economy of the protocol, the time service has the sole role of notifying a process that a timeout is expired. This is done by gathering timeout expiration notification requests by means of *Event Listeners*, objects with which a process notifies the time service about its own interest for a determinate event. In the scope of the protocol, the only event a process is interested in is timeout expiration, and interest towards notification of a timeout expiration is notified to the time service by means of a `TimeoutListener` object, shown in Figure 4.1.2

`clockedTimeout` is the `originalTimeout` adjusted to the time service's own clock. This is needed in case the owner process' clock is out of alignment with the time service's clock. In this case values of the two fields differ, while in case the two clocks are perfectly aligned, the value is the same. The `serviceTstamp` field contains a timestamp of when the `timeoutListener` has been sent to the time service, and is used when calculating the `clockedTimeout`. All other fields are self explanatory. The `TimeoutListener` class provides, besides methods to set and get values for various fields of the object, a method to notify the owner of expiration of a timeout.
When a `TimeoutListener` object is received by the time service, it is inserted by the main thread in a linked list, `listeners`, containing all `TimeoutListener`s received. Another thread, inside the time service, "ticks" the time by sleeping for a minimum amount of time and, on wake up, matching the current time against the `clockedTimeout` field of each `TimeoutListener` in the list. Timeout expiration is triggered by the current time being equal or bigger than the `clockedTimeout` field of an element. In this case, the time service notifies the owner by calling the `expirationNotification()` method that, on the owner's side, triggers the recovery thread.

4.1.3 QoS Monitoring Sub-service

Reliable Multicast layer is the bottom-most layer in the architecture, and at this level the QoS Monitoring Sub-service is mainly concerned on monitoring the underlying network reliability and performances.
Network QoS monitoring is one of today's most challenging task. Heterogeneity of network's nature in terms of network traffic and instability due to network protection technologies such as firewalls make it impossible to determine and even hard to estimate. All these factors lead to a sever of lack of standards in measurements.
Use of a multicast monitoring tool could be an approach to such a challenging task. For this purpose, a few tools for multicast trees monitoring have been revised. The most promising is the *Multicast Quality Monitor* (MQM) [22], that attempt to monitor QoS across a multicast group by using a combination of pinging to measure RTT (Round Trip Time) between nodes and the RTP (Real-time Transport Protocol) [21] to measure jitter and packet loss (given by RTP). Both these techniques do this by using separate measurement nodes on the group.
An alternative to the use of an external tool could be provided by the *IETF IP Performance Metrics*, which tries to develop a standard metrics that can be applied to the quality, performance, and reliability of Internet data delivery services.

4.2 External services

External services subpackage provide the protocol with capabilities disjoint from the ones proper of the protocol but even dough somewhat necessary for correct execution. In the current release, it contains two main services: the *Group Manager* realizes a basic group abstraction, while the *Violation Detector* monitors the layer's protocol. It's important to note that this subpackage contains services that might or might not be used in the context of the protocol. Both the services are described in details in the following paragraphs.

Group Manager: The Group Manager realizes a very basic group management protocol. Capabilities of this protocol, at the moment, include group formation, addition of new members and provision of references for group members. In this release, to form a group the first member starts an RMI registry on an remote or local host and regster a `RemoteGroupManager` service on it before adding itself as a member of the group. Subsequent members, then, are passed in initialization phase a couple `(hostName, port)` that uniquely identifies the machine hosting the RMI registry. Once the group is properly set up and ready to multicast, the RMI registry plays the important role of providing the multicast operation with a unique ID and provide group members with every information about the group itself by giving a `GroupRef` object upon request.
The `ExtServices` subpackage exports a `GroupManager` interface that provides APIs for the group management protocol. Its structure is shown in Figure 4.2.

This service has been included because protocol execution implies a group to be formed and ready to multicast, while at the moment it is not currently working with any group management protocol.

```
public interface GroupRef extends java.rmi.Remote {
    public localGroup createGroup(String name);
    public void addMember(Member newMember);
    public int getSize();
    public String getName();
    public LinkedList getGroup();
    public Member getMember(int index);
    public int setPort(int port);
    public int getPort();
    public int getMemberPort(Member member, int type);
    public int setMcastParams(int rho, double eta);
    public int getRho();
    public double getEta();
    public long getTimeStamp();
    public Member getSender();
    public boolean setSender(String newSender, int port);
    public long getMcastID();
    public LinkedList getReceivers();
}
```

Figure 16: `groupManager`'s APIs

Violation detector: The Violation Detector has the possibly important role of monitoring that a service does not violate some service level eventually specified in an SLA. Even if at this stage the protocol does not make use of this external service, it is important to emphasize that this type of monitoring highly differs from monitoring realized with the QoS Monitoring Sub-service and to understand why it has been defined as an external service rather than part of the QoS Monitoring Sub-service. Monitoring obtained by the QoS Monitoring Sub-service is aimed to validate guarantees given by the Negotiation Sub-service to the Core Protocol Sub-service. From this definition is easy to understand how this monitoring is towards the internals of the protocol itself, and how it does not affect QoS service level defined inside an SLA, that could eventually be target of eventual violations detected by the Violation Detector. Therefore, this service can be seen closer to the TAPAS general monitoring service rather than the protocol's own one. Again, it is important to note that this service might even not appear for some layers, since the referring SLA could not include any clause regarding the use of the service.
The Violation Detector is defined by the `ViolationDetector` interface, implemented by the `ViolationDetectorImpl` class. This class is composed by a thread that basically monitors the Core Protocol Sub-system, doing nothing when the service provided by the sub-system lies within certain thresholds specified inside an SLA and taking some action in case the service offered goes beyond the one specified in the SLA. Since this service requires a concrete integration with other components of the TAPAS platform, at this stage it is still not clear what APIs this service should offer.

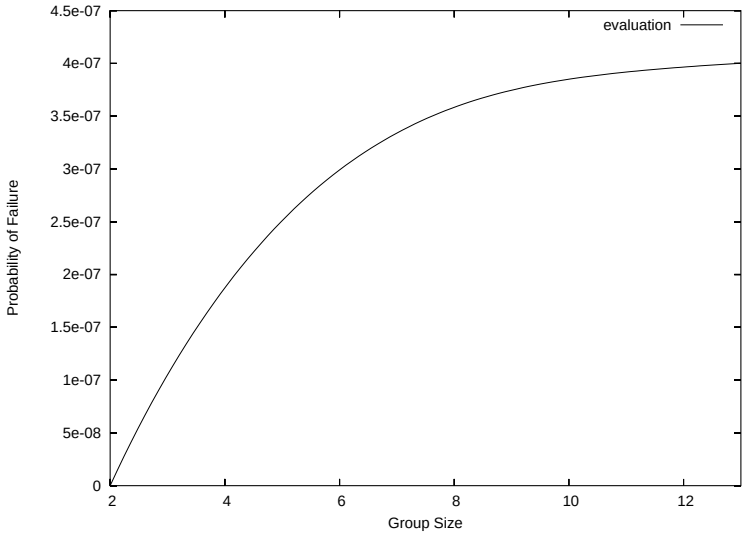

Figure 17: Protocol's failure probability

5 QoS Validation

5.1 Validation of reliability guarantees

We here evaluate the probability that the agreement will not be met. Evaluation has been made by deriving an expression for failure probability and evaluated it. Results of evaluation are shown in Figure 5.1, while details on derivation of the expression can be found in appendix C. In appendix C, we name the probability of failure as f_n. If we name the probability of success A, than probability of failure $f_n = (1 - A)$. Figure 5.1 shows the probability of failure for the protocol, and shows how the protocol is very unlikely to fail. For a group size $n = 2$, probability of failure is zero. In this case, in fact, the group is composed by an originator and a single receiver that can either receive the message, in which case the protocol is successful, or loose the message, in which case the protocol does not start according to the definition of reliable multicast. When $n > 2$, probability of failure starts to grow and seems to stabilize to a factor of 10^{-7}.

5.2 Validation of latency delay guarantees

The protocol performance has been simulated for a variety of parameter values. Reliability has been evaluated too, and results are presented here. Each simulation experiment consists of 100 independent runs of the protocol, using the same parameter values but different random number streams. The probability r_D is estimated as the fraction of the 100 runs for which all destinations receive m within time D. Similarly, u_S is estimated as the fraction of the 100 runs for which all remaining *operative* destinations receive m within time S after its arrival at a given *operative* process. The following scenarios were considered:

1. *No crashes.* All processes remain operative throughout.

2. *Originator crashes.* The originator crashes after completing the broadcast of copy number 0. Due to message losses, some receivers may not receive m directly from the originator.

3. *Originator crashes with a small set of direct receivers.* The originator crashes while broadcasting copy number 0, such that only a small set of processes directly receive m. The size of that set, called the *direct receivers*, is varied.

In all simulations, message transfer times are distributed exponentially with mean $d = 1$; the message loss probability is $q = 0.05$; the group size is $n = 50$; the level of certainty is $\alpha = 0.99$, resulting in $\eta = 4.6$.

As well as the performance metrics mentioned already, the simulations count the total number of broadcasts performed during each run; these counts, denoted as *bcasts*, are averaged over the 100 runs. Five groups of experiments were performed:

In group 1, scenarios 1 and 2 were implemented, with $\omega = 0$ and $\rho = 1$.

In group 2, scenario 3 holds, again with $\omega = 0$ and $\rho = 1$; the number of direct receivers was: 1, 2, and 5.

Groups 3 and 4 are the same as 1 and 2 respectively, except that $\rho = 2$.

Group 5 is the same as 3, but with dynamically adaptive timeouts.

Figure 17 shows the estimated and observed probability of success, r_D, as a function of D, for group 1. When there is no crash, the approximation is an under-estimate throughout, because it ignores the possibility that receivers may time out and become broadcasters; the latter is not unlikely, since $\omega = 0$ (in fact, an average of more than 4 broadcasts were observed, instead of 2). When the originator is allowed to crash, the approximation is an over-estimate until receivers time out (at $\eta + \zeta$) and become broadcasters. Then it again becomes an under-estimate.

Figure 18 illustrates the results for group 2, where the originator crashes while attempting to broadcast copy number 0, and the number of direct receivers is quite small. The probability of success, u_S, is plotted against the relative delay, S (relative to the first receiver). As expected, the larger the number of direct receivers, the better the performance. The approximations generally under-estimate the probability of success, except when S is small and/or the number of direct receivers is 1. Then the observed under-performance is caused by the other processes being artificially prevented from receiving directly from the originator, whereas the approximation allows it.

Figures 19 and 20 represent groups 3 and 4 respectively, with $\rho = 2$. In figure 19, the probability of success, r_D, is plotted against the absolute latency, D. The behaviour of the approximations and observations is similar to that in Figure 17. The increased value of ρ improves the approximated probability of success considerably.

Figure 20 shows the probability of success, u_S, plotted against the relative delay, S (relative to the first receiver), when the originator crashes while attempting to broadcast copy number 0. Because the few direct receivers now make 3 broadcasts, u_S is closer to 1 for large values of S. The approximation is again an over-estimate when the number of direct receivers is 1 or 2, for the reasons mentioned above.

Consider the observed message traffic. When $\rho = 1$ and the originator remains operative, ideally there would be 2 broadcasts in total, whereas the

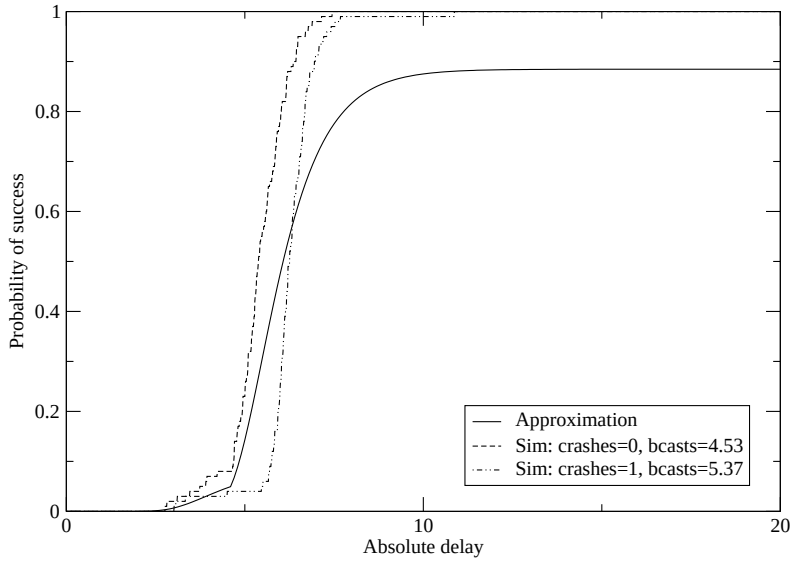

Figure 18: Group 1: r_D as a function of D; $\rho = 1$, $\omega = 0$

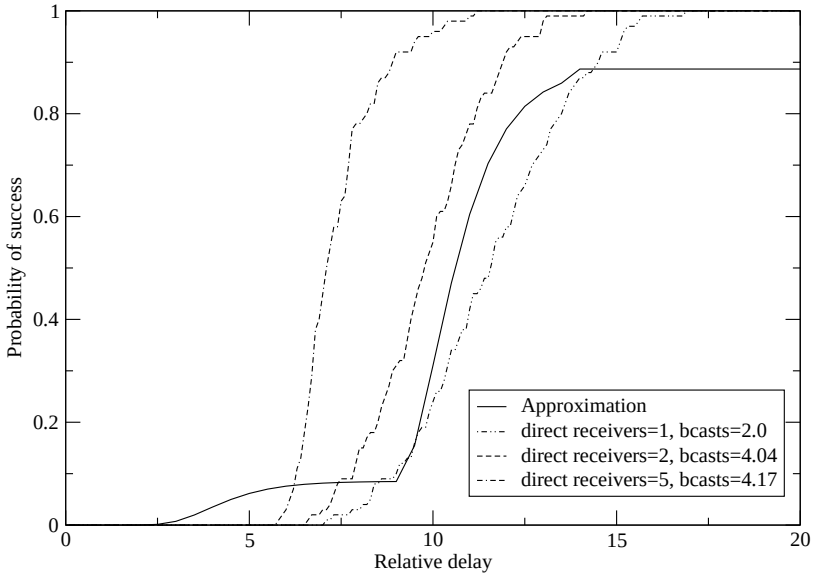

Figure 19: Group 2: u_S as a function of S; different numbers of direct receivers; $\rho = 1$, $\omega = 0$

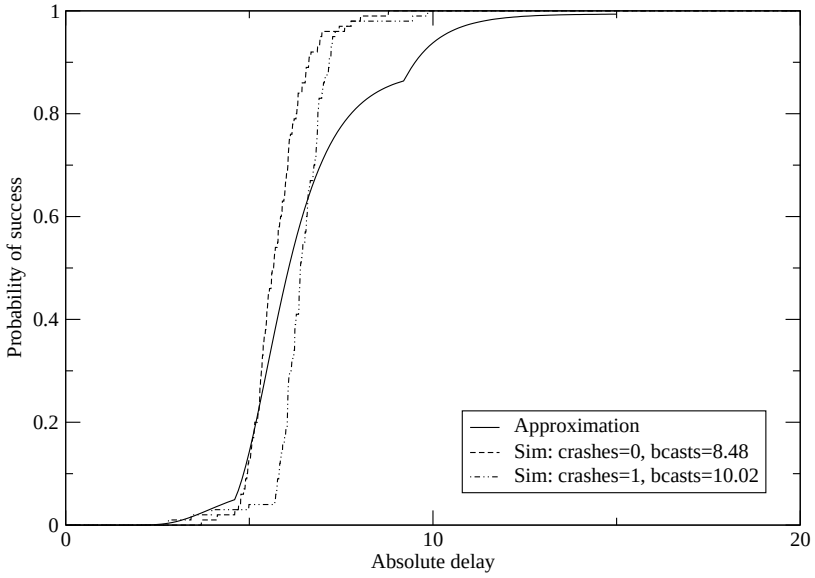

Figure 20: Group 3: r_D as a function of D; $\rho = 2$, $\omega = 0$

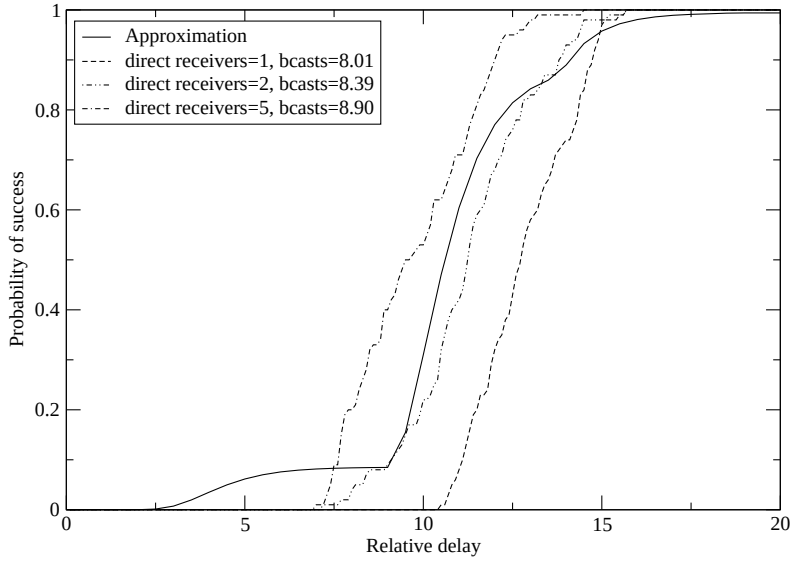

Figure 21: Group 4: u_S as a function of S; different numbers of direct receivers; $\rho = 2$, $\omega = 0$

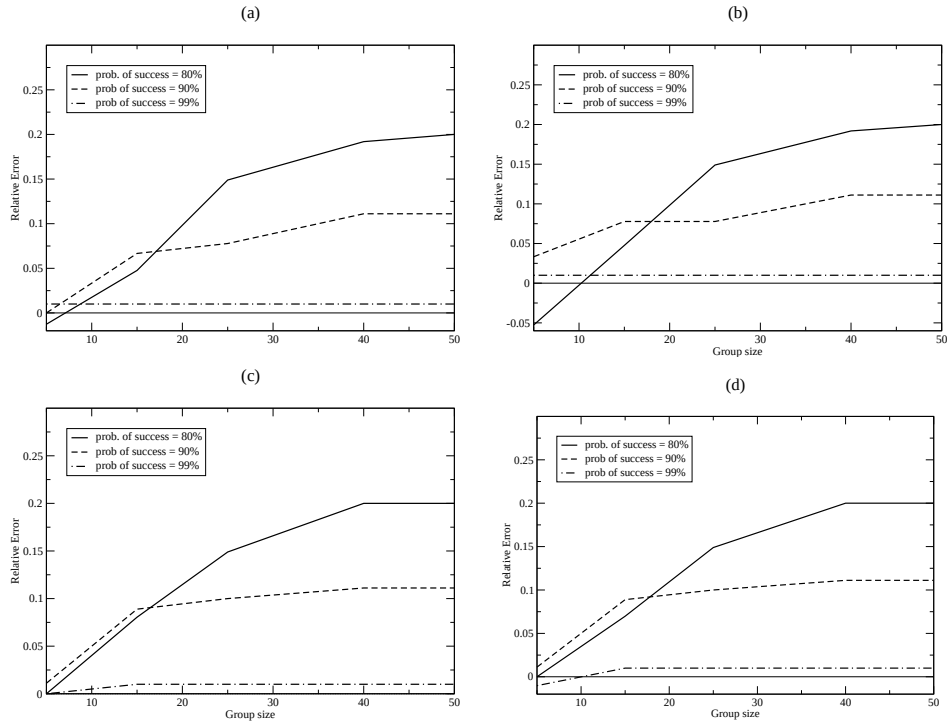

Figure 22: Relative error on simulation: on relative latency delay ((a) and (b)), and on absolute delay latency ((c) and (d)).

observed average is 4.53; when the originator crashes after making 1 broadcast, the ideal figure is 3 and the observed one is 5.37 (Figure 18). Similar ratios of ideal/observed number of broadcasts hold when $\rho = 2$ (Figure 20). Thus, the price paid for high reliability without dynamic adaptation is a 2 to 3-fold increase in message traffic compared to the unattainable ideal.

A further proof of how the approximations underestimate simulations is given by Figure 22. This figure shows the *relative error* of the approximation with respect to the simulation varying the group size. In the graphs, relative error is shown for different group sizes. In each graph we fix three success probabilities (e.g. 80%, 90% and 99%). For each of these, we take the time at which the approximation calculations reach such probability. We then see what probability of success we are able to reach in reality (i.e. in the simulation) and make it relative. These graphs basically show how probability of success guaranteed by means of approximation differs from probability of success truly reached in simulation over different group sizes. Positive errors here mean that approximation is really underestimating simulation, while negative errors mean that approximation is overestimating simulation.

Figures (a) and (b) show relative error on absolute latency delay and relative latency delay respectively. Here simulations have been conducted with the same set of parameters as Group 3 at the beginning of this section. Figures (c) and

Size	non-adaptive	adaptive	reduction
5	3.97	3.92	1.25%
15	4.73	4.43	6.34%
25	5.03	4.47	11.13%
40	6.90	5.65	18.11%
50	8.48	6.78	20.04%

Figure 23: Reduction of total number of broadcasts due to dynamic adaptation of timeout in simulation, *no crashes* scenario.

Size	non-adaptive	adaptive	reduction
5	4.21	4.06	3.56%
15	5.19	4.77	8.09%
25	6.77	5.78	14.62%
40	8.21	6.64	19.12%
50	10.02	7.91	21.05%

Figure 24: Reduction of total number of broadcasts due to dynamic adaptation of timeout, *originator crash* scenario.

(d), showing again relative error on absolute latency and relative latency delays respectively, are obtained by simulations where probability of packet loss q has been increased to 7.5%. These graphs clearly show that, despite the group size, the service achieved in reality is constantly better than the one we guarantee to the user, with peaks of 20%.

The effect of dynamically adaptive timeouts was also simulated. The results are shown in Figure 23 (in the *no crashes* scenario) and Figure 24 (in the *originator crash* scenario). What changes is the total number of broadcasts during the execution of the protocol. A reduction of 15% – 20% in the total number of broadcasts was observed.

6 Current Status and Integration Plan

6.1 Current status

As stated earlier, RMCast has been implemented in Java as a CORBA service. CORBA provides a software infrastructure allowing services to support applications that may be implemented in a number of different languages to be deployed in a heterogeneous environment. This relieves application developers from having to write their applications in a specific language targeted for specific platforms to use RMCast. We assume RMCast to be used by a variety of

application types beyond our requirements in the TAPAS project. For example, RMCast will benefit message dissemination schemes for real-time multimedia technologies in the support of applications such as online games, traditionally written in C or C++, where timeliness and reliability of message delivery play an important role in satisfying end user requirements. RMCast is currently a standalone service that is deployed on a per-node basis, with access provided to processes co-located on the same node via an Interface Definition Language (IDL) interface. IDL supports the description of services based on the object-oriented paradigm without the need to specify the target implementation language (separating interface from implementation). Section 4 contains the details of the APIs. In order to describe how the multicast service is used by developers, we now provide a brief explanation of the functionality provided via the IDL interfaces associated to RMCast (see Figure 25).
`RMGroupFactory` provides developers with the ability to instantiate multiple instances of the `RMGroup` interface. We assume that a developer would instantiate an `RMGroup` interface on a per-group basis and so provide access to a group on a per-node basis. Via `RMGroupFactory` a developer specifies the name of the group (useful for higher level services such as group management but not necessary for `RMCast`) and the port on which to expect incoming messages from the group. Once created, the `RMCast` method of `RMGroup` may be called to send a message. A message is in the form of a CORBA `any` data type. The `any` data type serves as a container for any data that can be described in IDL or for any IDL primitive type. This provides a flexible way of representing information whose type is unknown at compile time (expected in such a generic service). Together with the message, a list of target recipients is provided (associated to the membership of the group the multicast is sent to). This list may be derived from higher level services such as group membership for dynamic groups (membership changes throughout the lifetime of the group) or from a naming service when group membership is static (does not change throughout the lifetime of the group).

6.2 Integration Plan

Although RMCast has been implemented as a standalone service, within the context of the TAPAS project, we plan to integrate it within the JBOSS application server (the server chosen for the TAPAS QoS enabled application server platform). The JBOSS application server makes use of *JavaGroups* (JGroups) group communication system. We describe how RMCast can be used by JGroups, and the advantages of such an integration. This integration work will be completed by September 2004, to be ready as a part of demonstration of TAPAS platform.

6.2.1 JGroups in JBOSS

JBOSS provides clustering of application services to support load balancing and fail over. Clustering allows any one of a number of application servers to satisfy a client request and so may be described as scalable in the sense that increased numbers of client requests may be satisfied by providing additional resources in the form of the addition of application servers to a cluster. A load balancer assumes responsibility for attempting to distribute processing and communica-

```
module CoreProtocolService{
  struct groupMember{
    string memberID;
    string IPAddress;
    short portNumber;
  };

  typedef sequence<groupMember> targetDelivery;
  typedef sequence<any> deliveredMsg;

  interface RMGroup{
    void RMCast(in targetDelivery, in any msg);
    void RMDeliver(out deliveredMsg msg);
  };

  interface RMgroupFactory{
  }
```

Figure 25: IDL representing RMCast CORBA Service

tions activity across a cluster so that no single application server is overwhelmed. In JBOSS, a load balancer attempts to balance the load via a predetermined (e.g., round robin) approach to client request distribution amongst application servers in a cluster. An additional benefit of clustering is the ability to satisfy client requests given limited application server failures: client requests may be redirected to only correctly functioning application servers and away from application servers that are suspected to have failed. A group communications service is an important sub-system in clustering: providing cluster membership management (application servers joining and leaving a cluster) and informing the load balancer of such changes. This requires consensus protocols to realise the membership of a cluster that incorporate failure suspicion mechanisms to realise when application servers may have failed. JavaGroups (JGroups) currently supports these services for JBOSS clustering.
JGroups is a toolkit for constructing and managing protocols that provide multicast services for use in Java applications. JGroups provides an API that incorporates the notion of a channel for enabling group participants (geographically separated processes) to disseminate messages to all other members of a multicast group. A user creates a channel and then calls a connect method together with a parameter identifying the name of the group the user wishes to join. A number of properties associated to a channel are specified by the user at channel creation time. These properties relate to the delivery properties of messages and group membership functions. Properties are specified in a manner that reflects a protocol stack: protocols governing unreliable message dissemination located near the base of the stack and higher level functions associated to ordering and group management located near the top of the stack. For the purposes of clustering in JBOSS, a channel needs to be setup that provides a comprehensive protocol stack. Such a stack contains 11 layers, including UDP

at the lowest layer for accessing networking protocols with FD (failure suspicion) and GMS (group membership) at higher levels. Other layers, although not mentioned here, are integral to the success of the group communication services used by JBOSS but are not included here for clarity (the understanding required to identify an appropriate stack is not necessary for the purpose of describing an overall view of how JGroups is used in JBOSS).
The *HighAvailable Partition* (HAPartition) is responsible for supporting access to basic clustering information (e.g., cluster name, cluster membership) and allows state transfer and RPC primitives to be enacted by higher clustering related services. The HAPartition uses JGroups for enabling group communications in the support of such services and exhibits its interface via the MBean mechanism associated with the JMX approach to the modular system development associated with JBOSS. Although HAPartitioning uses JGroups, the clustering documentation associated to JBOSS insists that *"the JBoss clustering framework has been abstracted in such a way that it should be possible to plug-in different communication frameworks"*.

6.2.2 Integration of RMCast into JBOSS

Given that HAPartitioning may use other types of group communication subsystems, it should be possible to replace JGroups with our QoS enabled group communications service. As JGroups and JBOSS are open source projects, the tailoring of existing JBOSS and JGroups code allows the integration of our software incrementally. Our approach is to replace protocols from the lower JGroup protocol stack first, followed by the higher level JGroup protocol layers. In the first instance, the RMCast protocol we have implemented replaces the lower part of the JGroups protocol stack, and so provides JGroups with QoS enabled reliable multicast. This approach would require minor modification to the JGroups source code and allow the HAPartitioning software to remain unmodified (as it is still gaining group communications services via the JGroups channels API). A process of replacing JGroup layers may then be achieved systematically with QoS enabled protocol layers built by ourselves. Approaching integration in such a manner will allow the existing services of JGroups to be available (e.g., failure suspicion, ordering) and so provide JBOSS with a fully functioning group communication sub-system at all stages of development.
Figure 26 identifies the initial integration of QoS enabled group communications into JBOSS. The lower UDP protocol layer (shown in grey) is modified in such a way that the code associated to socket access is rewritten to access an RMGroup object. To ensure the availability of an RMGroup object the JGroups code associated to stack creation is altered. This required the instantiation of the altered UDP protocol layer to also create an instantiation of an RMGroup object via the RMGroupFactory. The RMCast CORBA service is installed on a per node basis and is accessed locally via standard CORBA RMI calls by the UDP layer. As mentioned previously, this phase of integration is transparent to HAPartitioning. All data that the UDP layer would usually send via sockets is now transported via our RMCast objects.

A benefit associated with our initial integration of RMCast into JGroups is the ability to satisfy the requirements of WAN cluster deployment (nodes are not co-located on the same LAN but are located on geographically separated nodes) in the absence of IP-Multicast. ISPs are under no obligation to

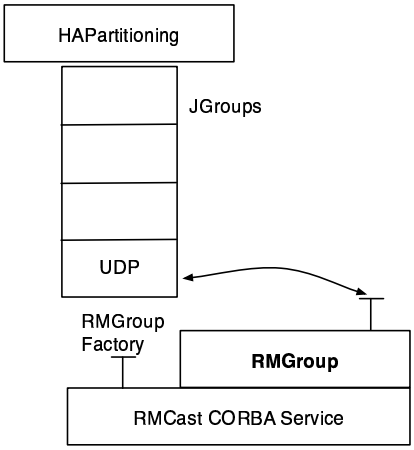

Figure 26: Initial integration of QoS enabled group communications service

support IP-Multicast. Therefore, JGroups provides lower stack layers that access TCP for WAN deployment in the absence of IP-Multicast. Maintaining TCP connections is a heavyweight approach to communications compared to UDP, although TCP does provide a higher degree of reliability (best effort) and FIFO message ordering over that provided by UDP. Unfortunately, the reliability and ordering of TCP is based on point-to-point messaging and requires JGroups to still instantiate a stack equivalent to when UDP is used as ordering and reliability of one-to-many communications is required. The ability to attain the QoS provided by RMCast means that our altered UDP layer may be used without resorting to TCP connections for WAN deployments that do not support IP-Multicast. Our approach requires less networking and processing resources compared to TCP. The integration phase will provide an appropriate testing platform for determining QoS parameters associated to different layers of the protocol stack and how negotiation between layers relating to QoS violations may alter such parameters. QoS parameters that relate to timeliness and reliability at the RMCast protocol layer (e.g., 90% of messages delivered within 50ms) will make it possible to provide higher layers with QoS. Failure suspicion is one such layer that will benefit from RMCast and provide adaptive QoS. In existing systems, once a failure is suspected an agreement protocol is initiated (by a member that suspects another member via some timeout). Such a protocol requires a subset of group members to agree on group membership, once agreement has been reached then application level processing of messages may continue. A problem in failure suspicion services is determining the appropriate timeout before a group members suspects another member of failing and initiating the agreement protocol: too short a timeout will result in unnecessary agreement protocol initialisations that have the effect of preventing application level processing from occurring whereas too long a timeout will result in failed members not being removed from the group (in the case of clustering this would have the effect of prolonging the issuing of client requests to what is actually a failed server). At the moment judging such timeouts is left to developers and such timeouts are static during runtime, not being capable to adapt to changes

in the environment (e.g., congestion of network). However, with RMCast such timeouts may be judged more appropriately and be adaptable at runtime given the negotiation capabilities of our QoS enabled architecture.

Conclusions

The protocol presented in this document offers a QoS-adaptive reliable multicast service that guarantees delivery of a message to all operative destinations despite sender or receiver crashes and message losses. The protocol also aims to eliminate unnecessary simultaneous broadcasts by multiple processes. The simulations confirm that the number of such broadcasts is not large. The expressions used in the QoS negotiations have deliberately been designed to be conservative and act as under-estimates. It has been shown by experimentation that they do indeed under-estimate the performance of the protocol, except in extreme cases which are very unlikely to occur in practice.
The basic (unordered) reliable multicast protocol has been implemented, as well as the negotiation mechanism needed to fulfill QoS requirements. APIs developed are general enough to easily be integrated inside any application server and to be run under any platform by means of the use of technologies such as CORBA, but leaves the possibility of the service be run as a stand alone application.. Ongoing work on implemented APIs include a real testing of the protocol, in order to have figures to be evaluated, and in depth study of efficient techniques for network-level QoS monitoring.
Future work will include development of protocols for more complete, sophisticated services such as FIFO and uniform ordering and integration of the service in the JBoss application server where it can be used for cluster management (e.g., for loadbalancing).

A Comparison of Models

The features of the probabilistic model are reiterated below, assuming a global clock (which is not accessible to processes).

- *Processing Delays*: Within a correct node, any task that is scheduled to be executed at time t, will be executed at $t + \pi$ where π is a random variable with some known distribution.
- *Storage Delays*: When a correct process initiates a storage request (for storing or retrieving of data) at time t, the request will be correctly processed at $t+\lambda$, where λ is a random variable with some known distribution.
- *Transmission Delays*: If a correct process i sends a message m to another correct process j at time t, then
  - m is delivered to j (i.e., m arrives at the buffer of j) with some probability $1 - q$ (m may be lost in transmission with probability q).
  - if m is not lost, it is delivered at $t + \delta$ where δ is a random variable with some known distribution.

If the distributions of π, λ, and δ are uniform with some known mean and $q = 0$, then the probabilistic model refers to the well-known synchronous model which permits upper bounds on π, λ, and δ to be determined with certainty; a violation of this bound is to be regarded as a failure of either the sending node or the receiving node. Thus, the synchronous model is a special case of the probabilistic model. This means that any probabilistic protocol designed for any given delay distribution and for a non-zero q should run correctly in a synchronous system when the delay distribution is uniform and $q = 0$. Conversely, if a problem is unsolvable in a synchronous system, then it cannot be solved in the probabilistic model. The asynchronous model considers the bounds on the delays π, λ, and δ to be finite; neither the bounds nor the delay distributions can be known with certainty. For example, any bound on delays, however judiciously deduced, is vulnerable to being violated with unknown probabilities. The probabilistic model, on the other hand, assigns probabilities or coverage to quantification of delay bounds.

The probabilistic model also differs in two ways from the two deterministic models over the treatment of message/packet loss. First, the losses are considered to be independent, though in reality they are likely to be correlated. This is abstracted for two reasons: it leads to a tractable performance analysis and the loss probability guaranteed by commercial ISPs is usually very small (about 1%).
Suppose that process i sends message m to process j a finite number of times, say k times. There is a small probability (q^k) that m is not delivered to j, whereas in the synchronous and the asynchronous models, the probability of a transmission failure with k, $k > 1$, attempts is assumed to be *nil* for some finite k. This assumption is often referred to as the *bounded degree omission* failures, and implies an underlying assumption that the losses are transient in nature and do not affect a flow between a given pair of processes permanently. The bound k is regarded to be known and unknown in the synchronous and asynchronous models respectively. Furthermore, in these deterministic models,

Models / Parameters	Synchronous	Asynchronous	Timed Asynchronous	Probabilistically Asynchronous
Bound on successive transmission losses, k	known	finite and known	finite and known	random variable on $[0, \infty]$
End-to-end delay for a 'sent' message, δ	has a known bound	has a finite and unknown bound	Has a known bound, if message not lost nor discarded by fail-aware	Random variable on $[0, \infty]$ with known distribution, if message not lost
Processing and Storage delays, π and σ	have known bounds	Have finite and unknown bounds	have finite and unknown bounds	random variables on $[0, \infty]$ with known distribution

Figure 27: Comparison of models

it is usual to abstract the redundant transmissions necessary to mask losses within the *send* operation[3] and to denote the over-all end-to-end transmission delay as δ. The *send* operation in the probabilistic model however refers to a single, non-redundant transmission. Thus, if q^k in the probabilistic model is taken to be zero for some k and the delay distributions be unknown and with finite support[4], then the probabilistic model becomes the asynchronous model. Figure 25 summarizes these observations on k and δ.

The timed asynchronous (TA) model [11] assumes a fail-aware service on top of an asynchronous CS. This service discards messages delivered by the asynchronous CS, if the messages are delayed by more than a threshold which is a fail-aware service parameter. Consequently, the CS of the TA model becomes much similar to that of the synchronous model (see Figure 25) except for a non-zero loss probability that can be much higher than q of the probabilistic model due to the filtering by the fail-aware service.

[3]Otherwise, correct processes cannot reliably send messages to each other, which is disallowed in both the deterministic models.

[4]A non-negative random variable ξ has a finite support distribution if $\mathcal{P}(\xi \leq x) = 1$ for some finite x.

B Analytical approximations for latency delays estimation

The probability, r_D, that all operative destinations receive at least one copy of a multicast message within a given interval of time, D, can be approximated by assuming that the originating process does not crash. This is a reasonable approximation because in practice processes crash rarely. Moreover, it will generally be a pessimistic approximation, since if the originator crashes at some point after broadcast 0 but before broadcast ρ, some of the processes that receive the last broadcast copy will make at least one broadcast themselves. Thus, the number of senders and hence the probability of success will increase. Of course it is possible that the originator crashes during broadcast 0, and no operative process receives any message; we consider the probability of that event to be negligible.

Let ξ be the random variable representing the execution time of a *send*(m) operation, i.e., the transmission time of a message from a given source to a given destination. The probability, $h(x)$, that such an operation *does not* succeed within time x, is equal to

$$h(x) = q + (1 - q)\mathcal{P}(\xi > x) \; , \tag{1}$$

where q is the probability that the message is lost. By definition, $h(x) = 1$ if $x \leq 0$. In the case of exponentially distributed transmission times (with mean d), the above expression becomes

$$h(x) = q + (1 - q)e^{-x/d} \; , \tag{2}$$

and $h(x) = 1$ for $x \leq 0$. Since the originator makes its kth broadcast at time $k\eta$ $(k = 0, 1, \ldots, \rho)$, the probability, g_D, that *a given destination* does not receive any of the $\rho + 1$ copies within time D, is given by

$$g_D = \prod_{k=0}^{\rho} h(D - k\eta) \; . \tag{3}$$

Hence, the probability, r_D, that every destination receives at least one copy of the message within an interval of length D is equal to

$$r_D = (1 - g_D)^{n-1} \; . \tag{4}$$

If some of the destinations have crashed, then (4) is an underestimate of the probability that all *operative* destinations receive at least one copy within time D. This is so because the term $(1 - g_D)$ would then be raised to a lower power, which would make the resulting probability larger.

A user requirement, stated in terms of a success probability R and latency D, is achievable if the probability evaluated by (4) satisfies $r_D \geq R$; otherwise it is not achievable.

B.1 Relative latency

Suppose now that at a given moment, t, a given process, p_i (different from the originator), receives copy number k of the message. Of interest is the probability,

$u_k(S)$, that all other processes will receive at least one copy of the message with relative latency S, i.e., before time $t + S$.

The implication of p_i receiving copy number k is that the originator has started broadcasting no later than at time $t - k\eta$ in the past, and has issued at least k broadcasts. Consider a given process, p_j, different from the originator and from p_i. The probability, $g_k(S)$, that p_j will not receive any of those k copies before time $t + S$ is no greater than

$$g_k(S) = \prod_{m=0}^{k} h(S + m\eta) \ , \tag{5}$$

where $h(x)$ is given by (1). In addition, if $k < \rho$, p_j may receive copies $k, k + 1, \ldots, \rho$ from p_i, in the event of the originator crashing. Those latter broadcasts would be issued at times $t + \eta + \omega + \zeta$, $t + 2\eta + \omega + \zeta$, ..., $t + (\rho - k + 1)\eta + \omega + \zeta$, assuming that no other process starts broadcasting. Since ζ is uniformly distributed on $(0, \eta)$, we can pessimistically replace ζ by η. The probability, $\tilde{g}_k(S)$, that p_j will not receive any of the messages from p_i before time $t + S$ is thus approximated by

$$\tilde{g}_k(S) = \prod_{m=1}^{\rho-k+1} h(S - (m + 1)\eta - \omega) \ , \tag{6}$$

where $\tilde{g}_\rho(S) = 1$ by definition; also, $h(x) = 1$ if $x \leq 0$. Thus, a pessimistic estimate for the conditional probability, $u_k(S)$, that all other processes will receive at least one copy of the message with relative latency S, given that a given process has received copy number k, is given by

$$u_k(S) = [1 - g_k(S)\tilde{g}_k(S)]^{n-2} \ . \tag{7}$$

A pessimistic estimate for the conditional probability, u_S, that all other processes will receive at least one copy of the message with relative latency S, given that a given process has received any copy, is obtained by taking the smallest of the above probabilities:

$$u_S = \min[u_0(S), u_1(S), \ldots, u_\rho(S)] \ . \tag{8}$$

This quantity may be used in deciding whether a user requirement, stated in terms of a success probability U and relative latency S, is achievable or not: the requirement is achievable if $u_S \geq U$. Intuitively, one would expect the minimum in the right-hand side of (8) to occur for $k = 0$, so that $u_S = u_0(S)$. Indeed, this has been the case in all examples evaluated.

Adaptive timeouts. Suppose that a user requirement stated in terms of U and S is achievable for some chosen ω and a given η. There may be scope for a dynamic adjustment of the parameter ω so as to minimize the message traffic rate. For example, suppose that the given process receives copy number k and, evaluating $g_k(S)$ according to (5), finds that

$$[1 - g_k(S)]^{n-2} > U \ . \tag{9}$$

That means that the user requirement can be achieved even if this process decides not to broadcast at all, i.e., sets its timeout parameter to $\omega = \infty$ (that would result in $\tilde{g}_k(S) = 1$). On the other hand, if (9) is not satisfied, then

$$[1 - g_k(S)\tilde{g}_k(S)]^{n-2} > U \tag{10}$$

must hold for the chosen ω because the user requirement was found achievable considering the smallest of $u_0(S)$, $u_1(S)$, ..., $u_\rho(S)$ in (8). Therefore the process may be able to set its ω to a larger value than the initially chosen one, while still satisfying the user requirement.

B.2 Heuristic adaptive timeouts

When a process first receives m with copy number k, ω can be set to ∞ if the expression (9) holds. If that expression does not hold but the requirement $u_S \geq U$ is achievable, then the best choice for ω is the largest feasible one:

$$\max\{\omega \mid u_k(S) \geq U\} . \tag{11}$$

However, that computation can be non-trivial. Moreover, a new value of ω needs to be computed whenever a new timeout is set for the same k, and for a value of S reduced by the time elapsed since receiving the first copy. Similarly, when the process receives the next copy, the value of S used in (11) should be the original target S reduced by the time elapsed since the m was first received. To avoid these complexities, we adopt a heuristic approach that simplifies the computation of adaptive timeouts.

Suppose that a receiver p_i receives m for the first time with $m.copy = k$.

If $k > 0$, p_i increases ω by $k\eta$. This is based on the assumption that the worst case assurance given for a given requirement $\{U, S\}$ relies only on copy number 0 having been received; but the receiver now knows that k additional broadcasts have already taken place.

If $k = 0$ and the receiver p_i receives copy 1 before the timeout $\eta + \omega$ expires, then ω is increased by η. The rationale for thus delaying the broadcast is to take advantage of other processes which time out on copy 1 and become broadcasters themselves.

C Analytical evaluations for reliability estimation

To calculate the probability of failure, two failure probabilities must be considered. One capturing the possibility that a single process fails before timeout expiration triggers that process to start transmitting, and another to capture the possibility for the originator to fail in the middle of each broadcast operation. Let's name then v the probability that a process fails before the timeout needed to start transmitting. If we assume this variable to follow an exponential distribution, then:

$$v = 1 - e^{-\tau\gamma} \tag{12}$$

where τ is the timeout itself and $\frac{1}{\gamma}$ is the mean time transmission failure (mttf). Let's also name β the probability that the originator fails inside each broadcast, after sending to k receivers. [5]
The scenario here is the one in which the originator starts to broadcast a message and crashes inside that broadcast operation after sending the message to k receivers. Since a message can be lost by the communication subsystem with probability $q > 0$, it is not sure that all k receivers will receive the message. It is reasonable, then, to assume that j receivers out of k will receive the message. These now can either fail altogether, causing the protocol to immediately fail, or at most $j-1$ of them fail and the j^{th} becomes the new leader and starts over the communication process with a system of $n-j$ processes, being subject to the same failure probabilities as at the beginning. The mathematical formula to describe this probability of failure is then, if we name f_n the probability that the protocol fails on a system with n processes:

$$f_n = \sum_{k=1}^{n-1} \left((1-\beta)^{k-1} \beta \sum_{j=1}^{k} \left(\binom{k}{j} (1-q)^j q^{k-j} (v^j + j v^{j-1}(1-v) f_{n-j}) \right) \right) \tag{13}$$

This formula is basic a composition of all possible probabilities that lead to protocol to fail, and needs an explanation:

$$f_n = \sum_{k=1}^{n-1} \left(\underbrace{(1-\beta)^{k-1}\beta}_{a} \sum_{j=1}^{k} \left(\underbrace{\binom{k}{j} (1-q)^j q^{k-j}}_{b} (\underbrace{v^j}_{c} + \underbrace{j v^{j-1}(1-v)}_{d} \underbrace{f_{n-j}}_{e}) \right) \right)$$

As said before, in this scenario the sender crashes in the middle of a broadcast, sending the message to k receivers out of $n-1$. This is captured by (a) in the expression, and the outer sum captures the fact that k can vary from 1 to $n-1$. Of these k receivers, only j receive the message, since there is a packet loss probability $q > 0$. (b) in the expression captures all possible combinations of choosing j receivers out of k, with j successful receptions and $k-j$ failing ones. Again, the outer sum captures the fact that truly receiving processes can go up to k. Once the message is received by j processes, each of these can either

[5] The broadcast operation is realized by sending the message to $n-1$ receivers, where n is the group size and is such that $n \geq 2$.

all crash before timeout's expiration or $j-1$ can crash leaving only one survivor process, that will go on with the multicast operation on a system with $n-j$ processes. Again, same assumptions about failing apply for this new system, and (e) captures the failing probabilities on a system with $n-j$ processes.
This formula slightly overestimates probability of failure, since considers only one process surviving to eventual crash in (d) and (e). Of course, more than one process could survive the crash in (d), and to capture this we should substitute (d) with the following:

$$(c) + \underbrace{\sum_{m=1}^{j-1} \binom{j}{m} (1-v)^m v^{j-m}}_{d} \underbrace{(f_{n-j+m})^m}_{e}$$

This new bit shows the possibility that more than one processes survive the probability to fail before the timeout (i.e. v) by including allowing m processes ($1 \leq m \leq j-1$) rather than one, and this captured by the initial sum. Internal binomial describes then all ways in which we can choose m processes out of j, of which m succeeding and $j-m$ failing. After this, we will have a system with $n-j+m$ processes with m processes firstly transmitting (senior run), so the probability of failure f_{n-j+m} is powered to m.

References

[1] Miley M. Reinventing Business: Application Service Providers. ORACLE Magazine, December 2000, pp. 48-52.

[2] Gibbens R. *et. al.* Fixed Point Models for the end-to-end performance analysis of IP Networks. In *Proceedings of the thirteenth International Teletraffic Congress Specialist Seminar: IP Traffic Measurement Modelling and Management*, September 2000, Monterrey, USA.

[3] Birman K and Joseph T. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1): 47-76, February, 1987.

[4] Hadzilacos V. and Toueg S. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, (Ed.) S Mullender, Addison-Wesley, 1993, pp. 97-146.

[5] Guerraoui R. Revisiting the relationship between Non-blocking Atomic Commitment and Consensus. In *Proceedings of the Ninth International Workshop on Distributed Algorithms*, Springer-Verlag, September 1995.

[6] F. Cristian. Synchronous and Asynchronous Group Communication. *Communications of the ACM*, 39(4):88-97, April 1996.

[7] Kopetz H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997, ISBN 0-7923-9894-7

[8] Aguilera M.K., Le Lann G., and Toueg S. On the Impact of Fast Failure Detectors on Real-Time Fault-Tolerant Systems. *16th International Symposium on Distributed Computing (DISC)*, Toulouse, France, October 28-30, 2002, pp. 354-370.

[9] Amir Y., Dolev, D., Kramer, S. and Malki, D. Membership Algorithm for Multicast Communication Groups. Proceedings of the *6th International Workshop on Distributed Algorithms*, pp 292-312, November 1992.

[10] Birman K., *et.al.* Bimodal Multicast. *ACM ToCS*, **17**(2), May 1999: 41-88.

[11] Cristian F and Fetzer C. The Timed Asynchronous Distributed System Model, In *IEEE Transactions on Parallel and Distributed Systems*, 10 (6): 642-57, June 1999.

[12] K. Birman, A. Schiper, and P. Stephenson,"Lightweight Causal and Atomic Group Multicast", ACM Transactions On Computer Systems, Vol. 9, No. 3, August 1991, pp. 272-314.

[13] Chandra T. D. and Toueg S. Unreliable Failure Detectors for Reliable Distributed Systems. *JACM*, **43**(2), pp. 225 - 267, March 1996.

[14] Cristian F. Synchronous and Asynchronous Group Communication. *Communications of the ACM*, **39**(4), pp. 88-97, April 1996.

[15] Eugster P.T., *et.al.* Lightweight Probabilistic Broadcast. *IEEE International Conference on Dependable Systems and Networks* (DSN01), 2001.

[16] Ezhilchelvan P.D., Shrivastava S.K., and Little M.C. A Model and Architecture for Conducting Hierarchically structured Auctions, *The fourth International IEEE Symposium on Object oriented Real-time Computing (ISORC)* May 2-4, 2001, Magdeburg, Germany, pp. 129-38.

[17] Floyd, S. Jacobson V. Liu C., McCanne S., and Zhang L. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, **5**(6): 784-803, December 1997.

[18] Hadzilacos V. and Toueg S. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, (Ed.) S Mullender, Addison-Wesley, 1993, pp. 97-146.

[19] Kopetz H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers. 1997.

[20] Laprie J.-C. Editor, *Dependability: Basic Concepts and Terminology*, Vol. 5 of *Dependable Computing and Fault-Tolerant Systems*. Vienna, Austria, Springer-Verlag, 1992.

[21] .Schulzrinne H. *et al.* RTP: A Transport Protocol for Real-Time Applications, RFC 1889, IETF, January 1996.

[22] Dressler F. MQM - Multicast Quality Monitor. In *Proceedings of 10th International Conference on Telecommunication Systems, Modeling and Analysis (ICTSM10)*, vol. 2, Monterey, CA, USA, October 2002, pp. 671-678.

[23] Lin J.C. and Paul S. RMTP: A Reliable Multicast Transport Protocol. *INFOCOM*, San Francisco, March 1996, pp. 1414-24.

[24] Lin J-C. and Marzullo K. Directional Gossip: Gossip in a Wide Area Network. *European Dependable Computing Conference (EDCC)*, 1999, pp. 364-379.

[25] Miley M. Reinventing Business: Application Service Providers. Oracle Magazine, December 2000, pp. 48-52.

[26] Sun Q. and Sturman D. A gossip-based Reliable Multicast for Large-Scale High-Throughput Applications. *IEEE International Conference on Dependable Systems and Networks* (DSN00), New York, 2000.