School of Computing Science,
University of Newcastle upon Tyne

Design and Evaluation of a QoS-Adaptive System for Reliable Multicasting

Antonio Di Ferdinando, Paul Ezhilchelvan and Isi Mitrani

Technical Report Series

CS-TR-833

March 2004

Design and Evaluation of a QoS-Adaptive System for Reliable Multicasting

Antonio Di Ferdinando, Paul D Ezhilchelvan and Isi Mitrani

School of Computing Science
University of Newcastle NE1 7RU, UK
{antonio.di-ferdinando, paul.ezhilchelvan, isi.mitrani}@ncl.ac.uk

May 6, 2004

Abstract

This document presents and studies a QoS-adaptive system for reliably multicasting messages to all intended destinations, despite possible crashes of the sender and other processes, and communication failures. This is a part of our on-going work on building a QoS-adaptive group communication system to support *application service provisioning* (ASP) where it is typical to purchase network level services, subject to some *service level agreements* (SLA), from an Internet Service Provider (ISP). The reliable multicast protocol studied here is designed for QoS metrics such as absolute and relative latency distributions, and the probability of successful delivery, to be negotiated prior to multicast provisioning. Moreover, the protocol adapts its parameters dynamically in order to minimize the message traffic required to achieve the negotiated QoS metrics. The performance of the protocol is analyzed mathematically under simplifying assumptions. The accuracy of the approximations is evaluated by simulations.

1 Introduction

The Internet is increasingly being used by organizations for offering and procuring services. Business outsourcing and application service provisioning [23] are some obvious manifestations of this trend. Obviously, when services are being traded, the networked systems providers who 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. It therefore becomes essential that the underlying system be able to evaluate the feasibility of QoS provisioning prior to accepting an end-user's 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 generally be associated with a system; *latency* (or *timeliness*) and *reliability* are common ones and will be the focus of this paper. 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 as 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 even 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 predicting queueing delays, scheduling and processing latencies, and failure rates of OS and SS under a given load and operational environment. 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 also can 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 [16, 14] 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 paper that the CS is provided by an ISP together

with well-defined network performance guarantees encapsulated in what the industry calls the *Service Level Agreements* (SLAs). Hosting distributed applications 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 [5, 18], consensus or atomic multicast, and non-blocking atomic commit [17]. Many GC systems have been built in the past [9]. Some assume the classical synchronous model, e.g., [13], and most others the asynchronous model or a variation of it; these are deterministic models. Observe however that the QoS guarantees offered by the ISPs (even to the high-end users) are not deterministic; rather they are *probabilistic* in nature. Note also that the network is not guaranteed to be 100% loss-less and 100% available; this means that a message may have to retransmitted and the reception time cannot be bounded with certainty but only in prababilistic terms. Assuming therefore a probabilistic model for interprocess communication delays and reliability, we focus on the construction of the most basic GC middleware service: (unordered) *reliable multicast*, *RM* for short. We address this problem comprehensively and the paper is structured accordingly.

Next section will present the assumptions and the *probabilistic model* that characterises an ISP supported communication system; it will also depict the three components which together make the *RM* system a QoS adaptive one. The *Service* component provides the *RM* service using an *RM* protocol designed with configurable parameters; the *QoS Negotiation* component sets these parameters appropriately to obtain the desired QoS levels and also to adapt to any QoS violations of the ISP which are observed by the third, *QoS Monitoring* component. Section 3 describes the *RM* protocol and Section 4 derives analytical expressions used by the *QoS Negotiation* component. These derivations make approximations for tractability reasons, which are chosen to have a bias towards underestimation so that the actual QoS offfered is better. The effectivesness of these approximations are studied through simulations in Section 5. Also examined are the adaptive nature of the protocol in minimising message overhead and reacting to CS-level QoS perturbs observed. Section 6 presents the related work and Section 7 the conclusions.

2 Assumptions, Communication Model and *RM* System Components

2.1 The Probabilistic Model

We consider a system of n, $n > 1$, distributed nodes that communicate using an ISP supported communication subsystem (CS). Each node hosts a distinct process p_i, $0 \leq i \leq n-1$. These processes coopearate with each

other as a group G for hosting an application (such as distributed e-auction). In line with the typical systems owned by service providers, the group size is known and small (n is in the order of 10s rather than 100s); also the process group is a closed one: processes 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 (i.e., ceases to be operative). A node (or a process) that does not crash is said to be correct. Each process has a primitive $send(m)$ using which it can send a message m to another process. The $send(m)$ is *successful* if m is deposited in the receive buffer of the destination process. We will assume that the processes of G are over-provisioned on computational and communication capacity. That is, queueing delays, processing delays, and scheduling delays can be assumed to be negligibly small compared to network delays. This means that a process can instantaneously receive a message which the communication subsystem deposits into its receive buffer, and the inter-process communication delay will be the message transmission delay over the network.

The communication subsystem (managed by an ISP) assures that when an operative process invokes $send(m)$ to send m to another operative process, then:

- the $send(m)$ operation is successful with a known probability $1 - q$, i.e., m is lost with probability q; and,
- if $send(m)$ is successful, the network transmission delay of m is an independent random variable with some known distribution.

We make the following simplifying assumption only for the purposes of simulations: the transmission delay distribution will be assumed to be exponential with mean d.

2.2 The RM Components and their Interaction

The QoS adaptive RM system lies on top of the ISP's managed network, and below the applications to be hosted (see Figure 1).

Figure 2 depicts the three components of the RM system:

1. The *RMService* component provides the RM service using a (fault-tolerant) protocol designed with configurable parameters; the choice of values for these parameters will influence the protocol behaviour and thus the QoS offered by the RM service. (These parameters can be regarded as *QoS control knobs*).

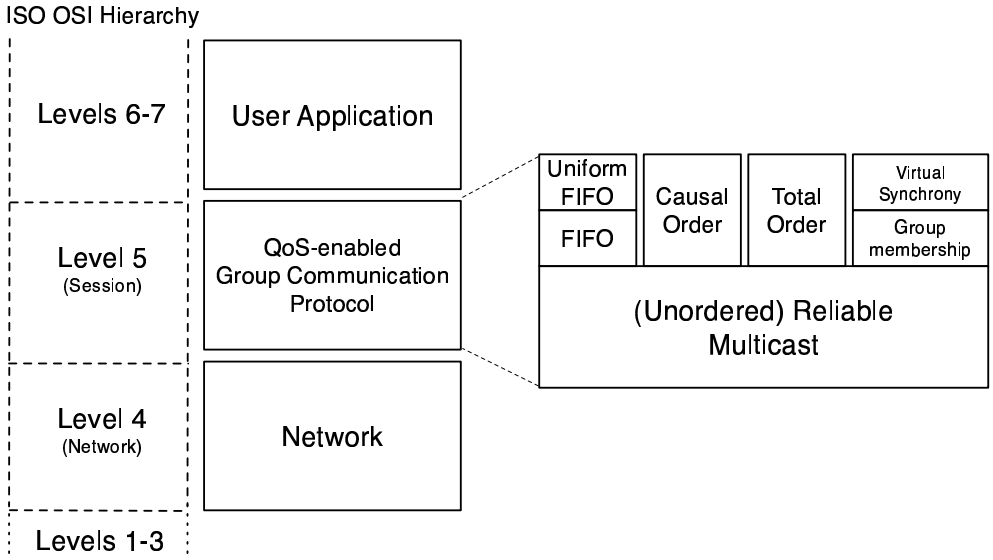

Figure 1: General architecture.

2. The responsibility for setting appropriate values to protocol parameters is upon the *QoS Negotiation* component. Setting of this parameter and the feasibility analysis to be carried out prior to accepting the an application request for a specified QoS will require that this component be equipped with algorithms or analytical expressions to evaluate the performance of *RM* service in terms of these parameters. Specifically, it should be able to evaluate the QoS metrics offered by the *RMService* 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 CS below (e.g., the level of redundancy for a given loss probability) and vice versa.

3. It is possible that the QoS guarantees agreed by the ISP are not met for a prolonged period. These violations can lead to the *RM* system being unable to meet its QoS obligations at run time. So, an essential requirement is to monitor the QoS offered by the ISP, and attempt to re-adapt the parameters of the *RM* protocol so that *RM* system maintains its QoS guarantees to the application. The monitoring and reporting activities are carried out by the *QoS Monitoring* component.

Referring to Figure 2, the interactions between the three modules of a subsystem can be summarised as below.

1. The *RMService* is issued a service request with some specified QoS metrics - shown in the figure as (1). The *QoS Negotiation* component

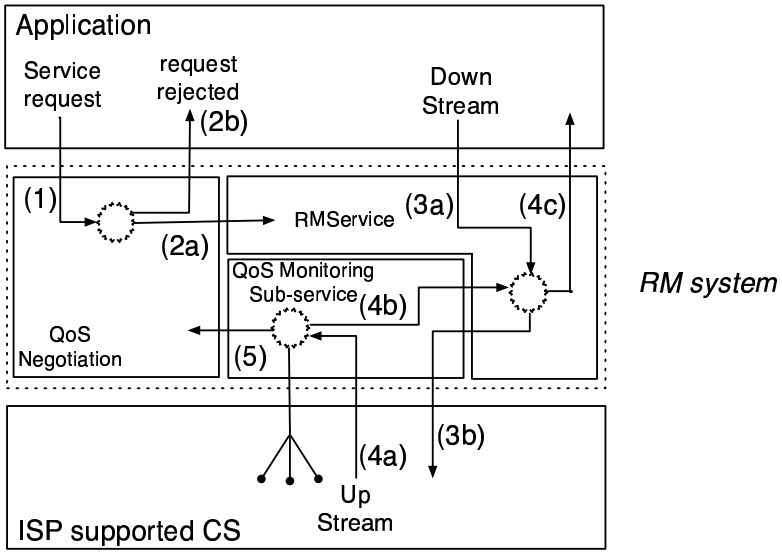

Figure 2: Component interaction

has the QoS guarantees of the CS which are currently in force. Based on these guarantees, it evaluates if the request can be met. If so, it accepts the request (2a) which is assumed here; otherwise, it rejects the request (2b).

2. The *QoS Negotiation* component sets the appropriate values for protocol parameters which the *RM* protocol should use to process messages related to the accepted request (3a).

3. Messages related to this request are processed by *RM* protocol and are passed down to *CS* for transmission via the *QoS Monitoring* component which tags the relevent down-stream messages so that the tagged ones can be monitored by the *ISP*. (3b)

4. Similarly, *RM* protocol receives the up-stream messages again via the *QoS Monitoring* component (4a and 4b), which collects data to compute the QoS metrics offered by the *ISP*, and delivers the message (4c).

5. The QoS metrics computed by the *QoS Monitoring* component are periodically sent to the *QoS Negotiation* component which in turn re-adapts the parameter values of *RM* protocol, if necessary and if possible, so that the QoS promised to the application is maintained (5).

3 Design and Description of a Reliable Multicast Protocol

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 an invocation of $RMcast(m)$ is delivered to a destination process by $RMDeliver(m)$. The protocol is designed with configurable parameters using which QoS offered can be set to the desired level. The QoS guarantees are probabilistic in nature and fall into two broad catagories: *reliability* and *latency*.

3.1 Specification of Protocol Guarantees

The protocol offers the following reliability guarantees (in probabilistic terms):

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

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 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 (with probability r_D); 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 (with probability u_S).

3.1.1 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 and changed to the desired effect. This aspect is utilised by the *QoS Negotiation* module to evaluate the feasibility of QoS support required from the $RMService$. 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 also possible at run time:

Adaptation for reduced message overhead. The evaluation of r_D and u_S involves taking approximations for reasons 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 thereby 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 has parameters which, when reset appropriately, can avoid failing to meet a given R, D or U, S due to unexpected QoS perturbs.

3.2 The Design Features.

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.

3.3 The Prototype Implementation

A first prototype of the protocol is under development. A working basic implementation of the reliable multicast protocol is in progress, and *Java* has been chosen as the referring programming language. Execution of a component is independent (perhaps synchronized) from execution of all other components. This is reflected by components' subpackage structure, to altogether form a package. The protocol's prototype exhibits interfaces via CORBA RPC, and the CS makes use of UDP datagrams. Communication amongst components has been realized using the *Java Messaging Service* (JMS). The interface for the *Negotiation* component contains primitives to handle QoS negotiation. The *RMService* component's interface exports primitives realizing the *RMCast(m)* and *RMDeliver()* operations. The *QoSMonitoring* component's interface exhibits primitives to measure performance metrics of a network. We have shown how our system adapts to QoS perturbs reported by the Monitoring component. QoS Monitoring is an active topic of research and there are many ways to implement the Monitoring component. Metric collection is central to QoS monitoring, which is concerned with gathering statistical information about the performance of a service. A good discussion of the advantages and limitations of existing techniques for metric collection is presented in [7]. The metric collector component (MeCo) [24] can be realised as one or more pieces of software possibly in combination with some hardware components. As an example of a monitoring system, we refer to EdgeMeter [25], a distributed meter system designed to monitor QoS of traffic of IP networks.

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.

RMcast(m)

(1) $m.originator \leftarrow i$; $m.SequenceNo \leftarrow seq\_number$;
(2) $m.copy \leftarrow 0$;
(3) **repeat**$(\rho + 1)$ times $\rightarrow$
(4) $\{$*broadcast(m)*; $wait(\eta)$; $m.copy \leftarrow m.copy + 1$;$\}$

Figure 3: Pseudo-code for $RMCast(m)$

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). An existing thread dies within η time after $leader_i(m) \neq i$ becomes true (lines 25, 28). When $\eta + \omega + \zeta > \eta$, the first thread dies by the time the next one is created. (If ω is set to be zero, then ζ can be chosen on $(\eta/2, \eta)$.) The thread pool in practical systems is not unbounded. So, our protocol makes judicial use of the available threads.

```
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
```

Figure 4: Pseudo-code for $RMDeliver()$

4 Analytical Estimations

4.1 Reliability Estimations

If the originator of a message m does not crash, then the only reason why some processes may not receive it, is losses in transmission. Since each transmission is lost with probability q, a given process will fail to receive all $\rho+1$ copies of m with probability $q^{\rho+1}$. Hence, the probability that all other processes receive at least one copy of m, i.e. the reliability of the protocol, r, is given by

$$r = (1 - q^{\rho+1})^{n-1} \ . \tag{1}$$

```
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 5: Pseudo-code for $Broadcaster(m)$

Clearly, this probability can be made as close to 1 as desired, by increasing ρ. Of course, the price paid for high reliability is increased latency and higher message traffic.

When crashes are taken into account, one should also consider the possibility that the agreement property may be violated. The following scenario may be realized: the originator crashes in the middle of the first broadcast, after executing only a few $send(m)$ commands; one or more of the destinations receive m and act upon it (e.g., become originators of new message(s), m'), but then they all crash before their timeouts expire and therefore fail to propagate m. In those circumstances, operative processes fail to receive m, while crashed ones receive it. Such a scenario may be termed a 'disagreement'. Intuitively, the occurrence of a disagreement is very unlikely, because it involves the conjunction of more than one event, each of which is unlikely. Nevertheless, it may be useful to estimate that small probability in terms of the processors crash characteristics.

There may be two kinds of breakdowns. Let v be the probability that a process crashes before its timeout expires. If the time-to-failure is distributed exponentially with mean $1/\gamma$, then v is given by

$$v = 1 - e^{-\gamma(\omega+2\eta)} \tag{2}$$

(pessimistically, ζ is assumed to take its largest possible value, η). This is typically a small number because γ is small.

Another breakdown mechanism operates while a process is broadcasting. Let β be the probability that it crashes just after a given $send(m)$ operation, independently of the others. Then we can write a recurrence relation for the probability, w_n, that a disagreement will occur in a group of size n.

$$w_n = \sum_{k=1}^{n-1} \left[(1-\beta)^{k-1} \beta \sum_{j=1}^{k} \binom{k}{j} (1-q)^j q^{k-j} [v^j + j v^{j-1}(1-v) w_{n-j}] \right] . \tag{3}$$

This relation quantifies the probabilities in the scenario outlined above: the originator crashes during the first broadcast, having executed k $send(m)$ operations; j of those messages are received at their destinations; then, either all j destination processes crash before their timeout expires, or one survives, becomes a broadcaster, but a disagreement occurs within the new group of size $n - j$ (the probability that more than one survive to become broadcasters, and still a disagreement occurs, is considered negligible).

The initial condition for the recurrences (3) is $w_2 = 0$, since a disagreement cannot occur with less than 3 nodes. When v and β are both small, the right hand side of (3) is on the order of $v\beta(1 - q)$.
As an example, Figure 6 shows the protocol's probability of failure for $\beta = 0.02$ (probability to fail inside each broadcast to 2%), allowing a failure every 100 hours on average ($\gamma = 100$ hours) and a timeout of $\omega + 2\eta = 5600$ millis. In this case, probability that a process crashes before its timeout expires is $v = 1.55 \times 10^{-5}$ and the overall probability of failure, increasing with the group size, seems to stabilize around a scale of 10^{-7}.

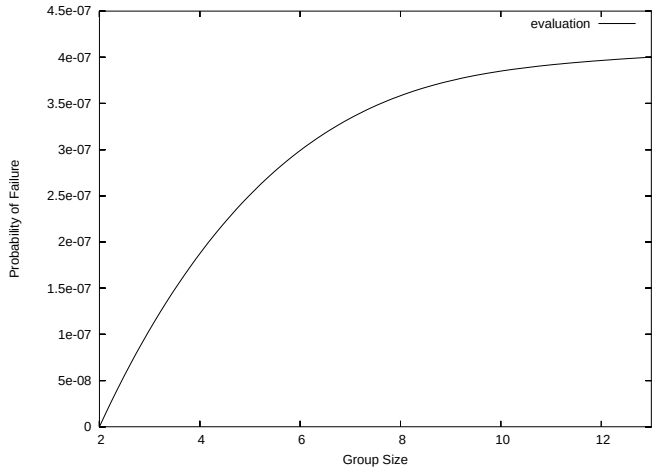

Figure 6: Protocol's failure probability, for $\beta = 0.02$, $\gamma = 100$ hours

4.2 Analytical Approximations for Latency 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

$$h(x) = q + (1 - q)\mathcal{P}(\xi > x) \; , \tag{4}$$

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

$$h(x) = q + (1 - q)e^{-x/d} \; , \tag{5}$$

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

$$g_D = \prod_{k=0}^{\rho} h(D - k\eta) \; . \tag{6}$$

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

$$r_D = (1 - g_D)^{n-1} \; . \tag{7}$$

If some of the destinations have crashed, then (7) 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 (7) satisfies $r_D \geq R$; otherwise it is not achievable.

4.3 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

$$g_k(S) = \prod_{m=0}^{k} h(S + m\eta) \ , \tag{8}$$

where $h(x)$ is given by (4). 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

$$\tilde{g}_k(S) = \prod_{m=1}^{\rho-k+1} h(S - (m+1)\eta - \omega) \ , \tag{9}$$

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

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

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:

$$u_S = \min[u_0(S), u_1(S), \ldots, u_\rho(S)] \ . \tag{11}$$

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 (11) to occur for $k = 0$, so that $u_S = u_0(S)$. Indeed, this has been the case in all examples evaluated.

4.4 Adaptive timeouts for reduced message overhead.

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 (8), finds that

$$[1 - g_k(S)]^{n-2} > U \ . \tag{12}$$

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 (12) is not satisfied, then

$$[1 - g_k(S)\tilde{g}_k(S)]^{n-2} > U \tag{13}$$

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 (11). 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.

Heuristic adaptive timeouts When a process first receives m with copy number k, ω can be set to ∞ if the expression (12) 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:

$$\max\{\omega \mid u_k(S) \geq U\} \ . \tag{14}$$

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 (14) 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.

4.5 Adaptation to QoS Perturbs.

If the *Monitoring* component observes the performance of CS to be slow, with the mean delay larger than guaranteed, then the *Negotiation* component of destination processes can set smaller values for the parameter for ω. This will have the effect of those destination processes that have received some copy of m timing out on the *broadcaster* sooner i.e., suspecting the *broadcaster* very prematurely (see line 19 in Figure 4); consequently, each timed-out destination process will initially try to act as the *broadcaster* and this will maximise the chances of a receiver that has not received any copy of m, to receive some copy of m earlier than if only one process had been acting as the *broadcaster* over a slowed-down network. If, on the other hand, the CS performs faster than guaranteed, the *Negotiation* modules of destination processes can set larger values for ω, which will have the opposite effect: only the *originator* of m (if operative) is very likely to broadcast m which is sufficient in better conditions and also reduces the message overhead. The effect of adapting ω this way to achieve the desired latency metrics are analysed in the next section.

5 Experimental Results

The protocol performance was simulated for a variety of parameter values, and the results are compared against the analytical estimations. 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 7 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

$\eta +$

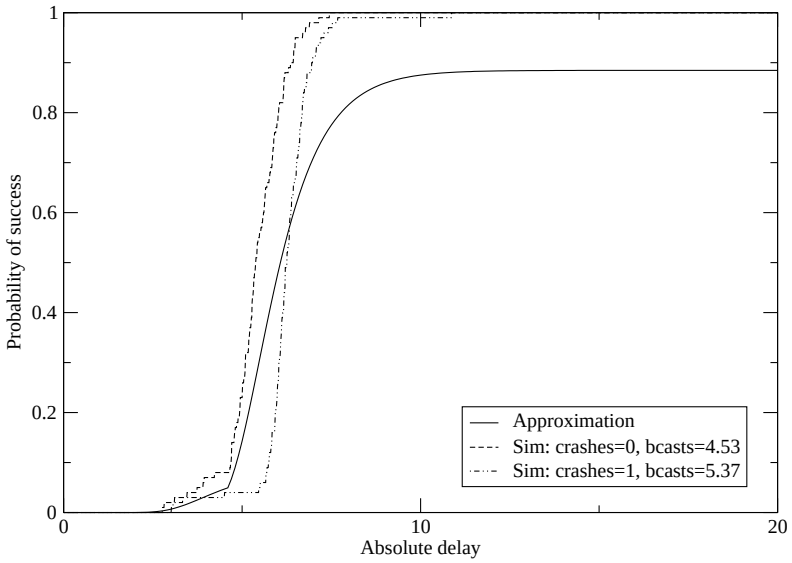

Figure 7: Group 1: r_D as a function of D; $\rho = 1$, $\omega = 0$

Figure 8 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

th

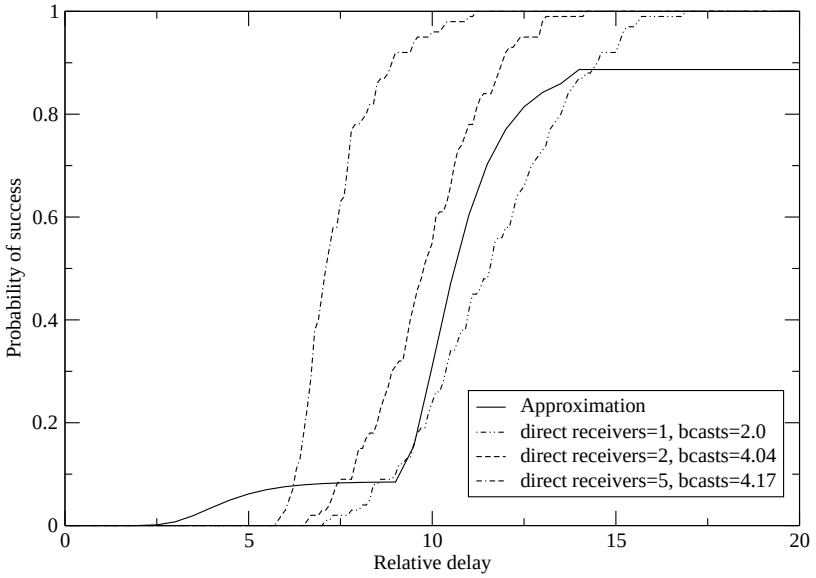

Figure 8: Group 2: u_S as a function of S; different numbers of direct receivers; $\rho = 1$, $\omega = 0$

Figures 9 and 10 represent groups 3 and 4 respectively, with $\rho = 2$. In figure 10, 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 2. The increased value of ρ improves the approximated probability of success considerably.

Figure 9 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 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 7). Similar ratios of ideal/observed number of broadcasts hold when $\rho = 2$ (figure 10). 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.

As a further proof for it, Figure 11 shows the *relative error* of the approximation with respect to the simulation varying the group size. The abscissa axis contains the group size, while the ordinates axis contains the

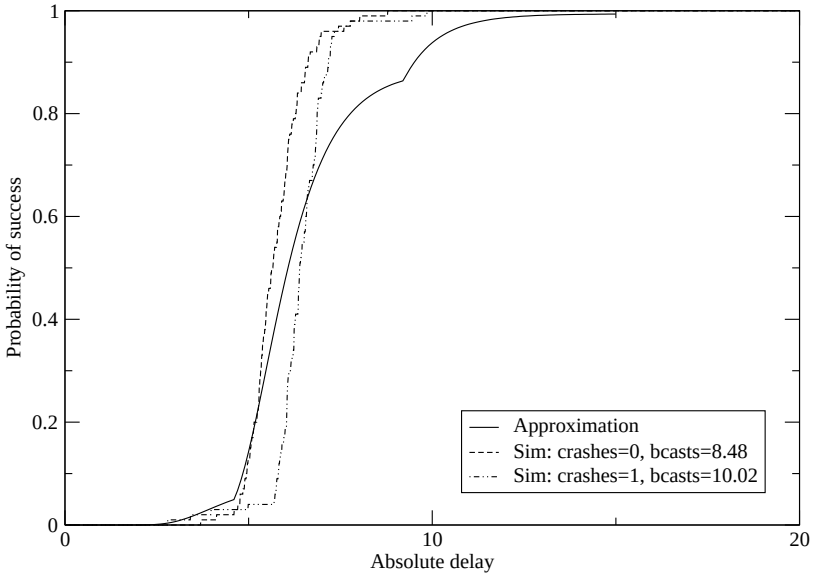

Figure 9: Group 3: r_D as a function of D; $\rho = 2$, $\omega = 0$

relative error. 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 (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%.

5.1 Effectiveness of Adaptive timeouts.

The effect of adaptive timeouts was also examined. Since adaptation takes place only after receiving the first copy, the probability of success remains unchanged. (Hence the graphs are not shown.) What does change is the total number of broadcasts during the execution of the protocol. A reduction

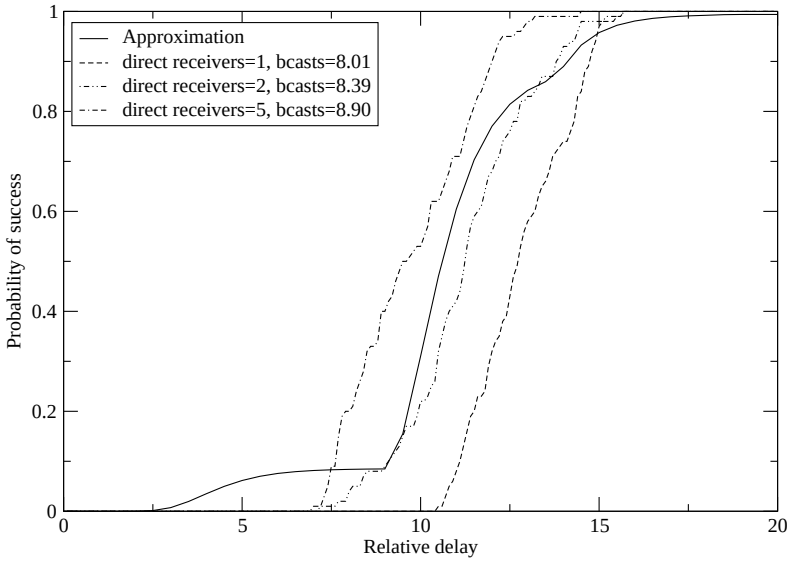

Figure 10: Group 4: u_S as a function of S; different numbers of direct receivers; $\rho = 2$, $\omega = 0$

of 15% – 20% in the total number of broadcasts was observed, as shown in figures 12 and 13.

5.2 Adaptation to Observed QoS Perturbs.

In the experiments described here, we assume that the *QoS Monitoring* component detects a 20% drop in the avarage delay d. That is, the CS is monitored to have become faster, or resumed normal after a period of poor performance. As mentioned in subsection 4.5, the adaptation will require increasing the value ω so that R and U promised to the application is maintained in the new context, and adaptation here will result in less message overhead.
In Figures 15 and 14 we chose 10 values of D and S from simulations running with the same experimental set-up as for group 3. The first column shows guaranteed success probability resulting from experiments where $\omega = 0$, the second column shows results for experiments where $\omega = 0.2$. The experiments highlighted an average reduction of total broadcasts number of 8-10%.

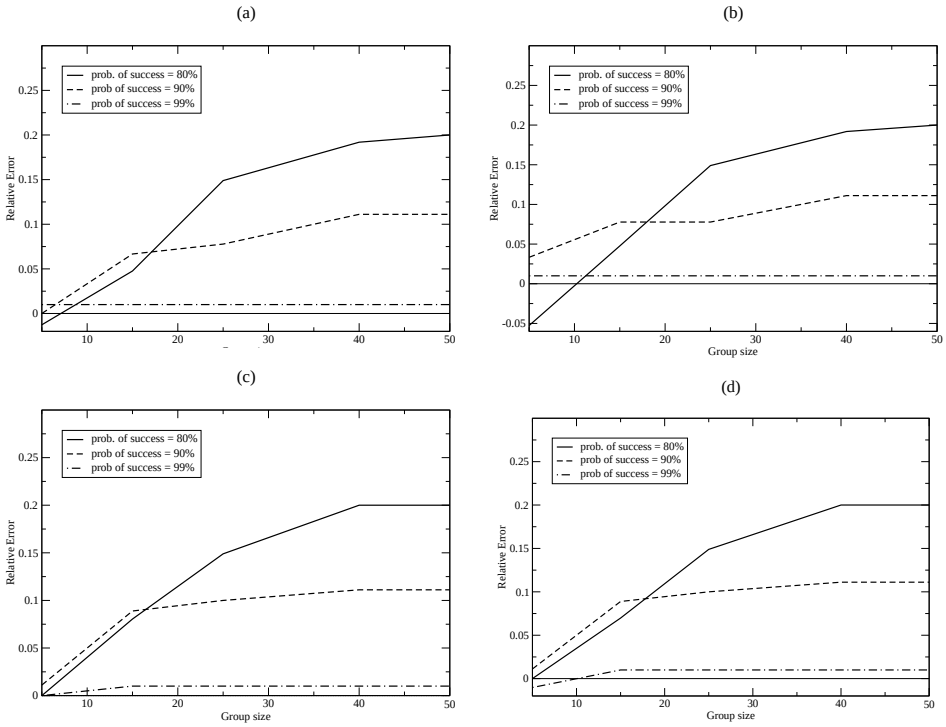

Figure 11: Relative error on simulation

6 Conclusions

The protocol presented in this document offers a QoS-adaptive reliable multicast service that guarantees delivery of a message to all or none of the correct destinations despite sender or receiver crashes and message losses. The protocol also aims to eliminate unnecessary broadcasts. 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.

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 12: Reduction of total number of broadcasts, *No Crash* 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 13: Reduction of total number of broadcasts, *Originator Crash* scenario

Latency		
D	$\omega = 0$	$\omega = 0.2$
2	0.0	0.0
2.5	0.0	0.03
3	0.0	0.04
3.5	0.0	0.05
4	0.04	0.07
4.5	0.04	0.1
5	0.05	0.27
5.5	0.06	0.86
6	0.29	0.99
6.5	0.7	1.0

Relative Latency		
S	$\omega = 0$	$\omega = 0.2$
2	0.0	0.0
2.5	0.0	0.03
3	0.0	0.04
3.5	0.0	0.05
4	0.04	0.07
4.5	0.04	0.1
5	0.05	0.29
5.5	0.07	0.86
6	0.31	0.99
6.5	0.74	1.0

Figure 14: Comparison of guaranteed success probability, *Originator Crash* scenario

Latency		
D	$\omega = 0$	$\omega = 0.2$
2	0.0	0.0
2.5	0.0	0.02
3	0.0	0.03
3.5	0.0	0.03
4	0.02	0.16
4.5	0.03	0.46
5	0.17	0.76
5.5	0.59	0.9
6	0.82	0.97
6.5	0.92	0.97

Relative Latency		
S	$\omega = 0$	$\omega = 0.2$
2	0.0	0.0
2.5	0.0	0.02
3	0.0	0.03
3.5	0.0	0.03
4	0.02	0.16
4.5	0.03	0.48
5	0.18	0.77
5.5	0.59	0.91
6	0.83	0.97
6.5	0.92	0.98

Figure 15: Comparison of guaranteed success probability, *No Crashes* scenario

References

[1] 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.

[2] 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.

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

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

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

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

[7] Cherkasova L., Fu Y., Tang W. and Vahdat A. "Measuring and Characterizing End-to-End Internet Service Performance", *ACM Transactions on Internet Technology*, 3(4), November 2003.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] Ezhilchelvan P. D. and Shrivastava S. K. "rel/REL: A Family of Reliable Multicast Protocols for Distributed Systems", *Distributed Systems Engineering*, 6:323 - 331, 1994.

[14] Firoiu V., Le Boudec J.-Y.,Towsley D.,Zhang Z.-L. "Theories and models for Internet quality of service", *Proceedings of the IEEE*, 90 (9)1565-1591, September 2002.

[15] 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.

[16] 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.

[17] 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.

[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, ISBN 0-7923-9894-7

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

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

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

[24] Molina-Jimenez C., Shrivastava S.K., Crowcroft J. and Gevros P., "On the Monitoring of Contractual Service Level Agreements", in *proceedings of First IEEE International Workshop on Electronic Contracting (WEC)*, July 6-9, 2004, San Diego, CA, USA.

[25] Pias M. and Wilbur S. "EdgeMeter: Distributed Network metering", In *Proceddings of the IEEE Openarch 2001 conference*, Anchorage, Alaska, USA, Apr. 2001.

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

[27] 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.