A Dynamic View-Oriented Group Communication Service*

Roberto De Prisco† Alan Fekete‡ Nancy Lynch† Alex Shvartsman§

Abstract

View-oriented group communication services are widely used for fault-tolerant distributed computing. For applications involving coherent data, it is important to know when a process has a *primary* view of the current group membership, usually defined as a view containing a majority out of a *static* universe of processes. For high availability in a system where processes can join and leave routinely, some researchers have suggested defining primary views *dynamically*, depending on having enough members in common with recent views.

We present a new formal automaton specification, DVS, for the safety guarantees made by a practical group communication service providing a dynamic notion of primary view. We demonstrate the value of DVS by showing both how it can be implemented and how it can be used in an application. First, we present a distributed algorithm based on a group membership algorithm of Lotem, Keidar and Dolev; our version integrates communication with the membership service, uses information from the application processes saying when a view has been prepared for computation by the application, and uses a static view-oriented service internally. We prove that this algorithm implements DVS. Second, we present an application algorithm that is a variant of an algorithm of Amir, Dolev, Keidar, Melliar-Smith and Moser, modified to use DVS instead of a static service. We prove that it implements a (non-group-oriented) totally-ordered-broadcast service.

1 Introduction

View-oriented group communication services have become important as building blocks for fault-tolerant distributed systems. Such a service enables application processes located at different nodes of a fault-prone distributed network to operate collectively as a group, using the service to multicast messages to all members of the group. Each such service is based on a *group membership service*, which provides each group member with a *view* of the group; a view includes a list of the processes that are members of the group. Messages sent by a process in one view are delivered only to processes in the membership of that view, and only when they have the same view. Within each view, the service offers guarantees about the order and reliability of message delivery. Examples of view-oriented group communication services are found in Isis [4], Transis [8], Totem [22], Newtop [11], Relacs [2], and Horus [24].

For maximum usefulness, system building blocks should have simple and precise specifications of their guaranteed behavior. Producing good specifications for view-oriented group communication services is difficult, because these services can be complicated, and because different such services provide different guarantees. Examples of specifications for group membership services and view-oriented group communication services appear in [3, 5, 6, 9, 14, 15, 21, 25, 26]. In [12], we presented a specification, VS, for a view-oriented group communication service. This specification consists of a state machine expressing safety requirements, plus a timed trace property expressing conditional performance and fault-tolerance requirements. We used this specification as the basis for proving the correctness of a complex totally-ordered-broadcast algorithm based on [17, 1].

The VS service produces arbitrary views, with arbitrary membership sets. However, in many applications of VS, especially those with strong data coherence requirements, the application processes perform significant computations only when they have a special type of view called a *primary view*. For example, a replicated database application might only perform a read or write operation within a primary view, in order to ensure that each read receives the result of the last preceding write, in some consistent order of the operations. In this setting, a primary view is typically defined to be one whose membership comprises a *majority* of the universe of processes, or more generally, a *quorum* in a predefined quorum set in which all pairs of quorums intersect. The intersection property permits information flow from any previous primary to a newly formed one.

Pre-defined quorum sets can yield efficient implementations in settings where the system configuration is relatively static. However, they work less well in settings where the configuration evolves over time, with processes joining and leaving the system. For such a setting, a *dynamic* notion of primary is needed, one that can change to conform with the system configuration. A dynamic notion of primary still needs to maintain some kind of intersection property, in order to permit enough information flow between successive primary views to achieve coherence. For example, each primary view might have to contain at least a majority of the processes in the previous primary view. Several *dynamic voting schemes* have been developed to define primaries adaptively [7, 10, 16, 18, 23].

In particular, Lotem, Keidar, and Dolev [18] have described an implementation of a group membership service that yields only primary views, according to a dynamic no-

*This research was supported by the following contracts: ARPA F19628-95-C-0118, AFOSR F49620-97-1-0337, NSF 9225124-CCR, and by a grant from the GTE Laboratories.

†MIT Laboratory for Computer Science, 545 Technology Square, NE43-365, Cambridge, MA 02139, USA.

‡Basser Department of Computer Science, Madsen Building F09, University of Sydney, NSW 2006, Australia.

§MIT Laboratory for Computer Science, 545 Technology Square, NE43-371, Cambridge, MA 02139, USA and Dept. of Computer Science and Eng., 191 Auditorium Rd., U-155, University of Connecticut, Storrs, CT 06269, USA.

PODC 98 Puerto Vallarta Mexico

tion of primary. An interesting feature of their work is that it points out various subtleties of implementing such a membership service in a distributed manner – subtleties involving different opinions by different processes about what is the previous primary view. These difficulties have led to errors in some of the past work on dynamic voting. The algorithm of [18] copes with these subtleties by maintaining information about a collection of primary views that "might be" the previous primary view. The service deals with group membership only, and not with communication. Lotem et al. prove that their protocol satisfies the following condition on system executions: any two (primary) views that occur in an execution are linked by a chain of views where for every consecutive pair of views in the chain, there is some process that "knows" it belongs to both views.

In this paper, we present a new formal automaton specification, DVS, for the safety guarantees made by a practical dynamic view-oriented group communication service. This service is inspired by the implementation of Lotem et al., but integrates communication with the group membership service.

We demonstrate the value of our DVS specification by showing both how it can be implemented and how it can be used in an application. First, we consider an implementation that is a variant of the group membership algorithm of Lotem et al.; our variant integrates communication with the membership service, uses "registration" information from the application processes saying when a view has been prepared for computation by the application, and uses a static view-oriented service (a version of VS) internally. We prove that this algorithm implements DVS, in the sense of trace inclusion. The proof uses a (single-valued) simulation relation and invariant assertions. The key to the proof is an invariant expressing a strong condition about nonempty intersections of views; the proof of this depends on relating a *local* check of *majority* intersection with known views to a *global* check of *nonempty* intersection with existing views.

Second, we consider an application algorithm that is a variant of an algorithm in [17, 1, 12], modified to use DVS instead of a static view-oriented service. The modified algorithm uses the registration capability to tell the DVS service that information has been successfully exchanged at the beginning of a new view. We show that it implements a (non-group-oriented) totally-ordered-broadcast service. This proof also uses a simulation relation and invariant assertions.

We have designed our DVS specification to express the guarantees that are useful in verifying correctness of applications that use the service. Among previous work, two different sorts of specifications for a primary group service are notable. Work by Ricciardi and others [26] is expressed in temporal logic on consistent cuts; the idea of their specification is that on any cut, there are no disjoint sets of processes such that each set is collectively aware of no members outside that set. Lotem et al. [18] use a property of an execution, which was previously defined by Cristian [6] for majority groups and that links any two (primary) views by a chain of views where every consecutive pair of views includes a process that "knows" it belongs to both views. As far as we know, these previous specifications have not been used to verify any applications running above them.

An important feature of our specification is our careful handling of the interface between the service and the application. When a new view starts, applications generally require some initial pre-processing to prepare for ordinary computation. For example, applications involving coherent data need to collect knowledge of changes from previous views, before allowing further activity. Our specification does not assume that information flows from one view to another merely because some process joins both. Instead, we treat the exchange of information as application-specific: We ask each application process to indicate, with a REGISTER event, when it has received all the needed information from other members of the new view v. When all members have registered v, the application has gathered all information it needs from previous views, and the service no longer needs to ensure intersection in membership between views before v and any subsequent ones that are formed. In contrast, in a specification based on a chain of views with common membership of successive pairs, one must have the application-level state exchange piggybacked on messages within the group management layer.

Our specification also omits some features of existing dynamic primary view management algorithms. For example, Isis [4] guarantees that processes that move together from one view to the next receive exactly the same messages in the first view. These properties are not needed to verify applications such as the one giving a totally-ordered broadcast. Of course there may be other applications that do require these stronger properties.

2 Mathematical foundations

We write λ for the empty sequence. If a is a sequence then $|a|$ denotes the length of a. If a is a sequence and $1 \leq i \leq j \leq |a|$ then $a(i)$ denotes the ith element of a and $a(i..j)$ denotes the subsequence $a(i), a(i+1), ..., a(j)$ of a. The *head* of a nonempty sequence a is $a(1)$. A sequence can be used as a queue: the *append* operation modifies the sequence by concatenating it with a new element and the *remove* operation modifies the sequence by deleting its head.

If a and b are sequences, a finite, then $a{+}b$ denotes the concatenation of a and b. We sometimes abuse this notation by letting a or b be a single element. We say that sequence a is a *prefix* of sequence b, written $a \leq b$, provided that there exists c such that $a{+}c = b$. A collection A of sequences is *consistent* provided that $a \leq b$ or $b \leq a$ for all $a, b \in A$. If A is a consistent collection of sequences, we define $lub(A)$ to be the minimum sequence b such that $a \leq b$ for all $a \in A$.

If S is a set, then $seqof(S)$ denotes the set of all finite sequences of elements of S. If $a \in seqof(S)$ and f is a partial function from S to T whose domain includes the set of all elements of S appearing in a, then $applytoall(f, a)$ denotes the sequence b such that $length(b) = length(a)$ and, for $i \leq length(b)$, $b(i) = f(a(i))$.

If S is a set, the notation $S_\perp$ refers to the set $S \cup \{\perp\}$. If R is a binary relation, then we define $dom(R)$, the *domain* of R, to be the set (without repetitions), of first elements of the ordered pairs comprising relation R. If f is a partial function from S to T, and $\langle s, t\rangle \in S \times T$, then $f \oplus \langle s, t\rangle$ is defined to be the partial function that is identical to f except that $f(s) = t$.

$\mathcal{P}$ denotes the universe of all processors,[1] and $\mathcal{M}$ the universe of all possible messages. $\mathcal{G}$ is a totally ordered set of identifiers used to distinguish views, with a distinguished least element g_0. A *view* $v = \langle g, P\rangle$ consists of a view identifier $g \in \mathcal{G}$ and a nonempty membership set $P \subseteq \mathcal{P}$; we write $v.id$ and $v.set$ to denote the view identifier and membership set components of v, respectively. $\mathcal{V}$ denotes the set of all views, and $v_0 = \langle g_0, P_0\rangle$ is a distinguished *initial view*.

We describe our services and algorithms using the I/O automaton model of Lynch and Tuttle [20] (without fairness).

[1] We use "processor" and "process" interchangeably.

Signature:

Input: VS-GPSND$(m)_p$, $m \in \mathcal{M}$, $p \in \mathcal{P}$
Internal: VS-CREATEVIEW(v), $v \in \mathcal{V}$
VS-ORDER(m, p, g), $m \in \mathcal{M}$, $p \in \mathcal{P}$, $g \in \mathcal{G}$

Output: VS-GPRCV$(m)_{p,q}$, $m \in \mathcal{M}$, $p, q \in \mathcal{P}$
VS-SAFE$(m)_{p,q}$, $m \in \mathcal{M}$, $p, q \in \mathcal{P}$,
VS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$

State:

$created \in 2^{\mathcal{V}}$, init $\{v_0\}$
for each $p \in \mathcal{P}$:
  $current\text{-}viewid[p] \in \mathcal{G}_\perp$, init g_0 if $p \in P_0$, $\perp$ else
for each $g \in \mathcal{G}$:
  $queue[g] \in seqof(\mathcal{M} \times \mathcal{P})$, init λ
for each $p \in \mathcal{P}$, $g \in \mathcal{G}$:
  $pending[p, g] \in seqof(\mathcal{M})$, init λ
  $next[p, g] \in \mathbf{N}^{>0}$, init 1
  $next\text{-}safe[p, g] \in \mathbf{N}^{>0}$, init 1

Transitions:

internal VS-CREATEVIEW(v)
  Pre: $\forall w \in created : v.id > w.id$
  Eff: $created := created \cup \{v\}$

output VS-NEWVIEW$(v)_p$
  Pre: $v \in created$
    $v.id > current\text{-}viewid[p]$
  Eff: $current\text{-}viewid[p] := v.id$

input VS-GPSND$(m)_p$
  Eff: if $current\text{-}viewid[p] \neq \perp$ then
    append m to $pending[p, current\text{-}viewid[p]]$

internal VS-ORDER(m, p, g)
  Pre: m is head of $pending[p, g]$
  Eff: remove head of $pending[p, g]$
    append $\langle m, p \rangle$ to $queue[g]$

output VS-GPRCV$(m)_{p,q}$, choose g
  Pre: $g = current\text{-}viewid[q]$
    $queue[g](next[q, g]) = \langle m, p \rangle$
  Eff: $next[q, g] := next[q, g] + 1$

output VS-SAFE$(m)_{p,q}$, choose g, P
  Pre: $g = current\text{-}viewid[q]$
    $\langle g, P \rangle \in created$
    $queue[g](next\text{-}safe[q, g]) = \langle m, p \rangle$
    for all $r \in P$:
      $next[r, g] > next\text{-}safe[q, g]$
  Eff: $next\text{-}safe[q, g] := next\text{-}safe[q, g] + 1$

Figure 1: VS (modified version)

The model and its proof methods are described in Chapter 8 of [19]. We use the term *refinement* to denote a single-valued simulation relation.

3 The VS specification

In this paper we use a modified version of the group communication service, VS, defined in [12], and we refer the reader to the informal service description in that paper. The original VS service assumes that every processor in the universe $\mathcal{P}$ is a member of the initial view. In our setting the initial view is defined to be the distinguished initial view v_0, and we modify the specification of VS to reflect this fact. The modified specification is given in Figure 1. The fact that VS allows views to be created only in order of view identifier is unimportant: weakening this requirement to allow out-of-order view creation would not change the external behavior, because VS-NEWVIEW actions are constrained to occur in order anyway.

Invariant 3.1 (VS)
If $v, v' \in created$ and $v.id = v'.id$, then $v = v'$.

4 The DVS specification

Our DVS specification differs from the VS specification in the following ways: (1) DVS-REGISTER actions allow a client of the service to notify the service that it is ready to begin operating in a new view. This information is recorded in new variables, $registered[g]$, $g \in \mathcal{G}$. (2) New variables, $attempted[g]$, $g \in \mathcal{G}$, are introduced to remember which views have been reported to each process. (These are used in the proofs.) Also, new derived variables are introduced to remember which views have been attempted or registered at some member or all members. (3) The action DVS-CREATEVIEW only creates primary components, whereas the VS-CREATEVIEW is unconstrained (except for increasing ids). The specification is given in Figure 2. In this specification, $\mathcal{M}_c \subseteq \mathcal{M}$ denotes the set of messages that clients may use for communication.

The most interesting part of the DVS specification is the transition definition for DVS-CREATEVIEW(v). The precondition specifies the properties that a view must satisfy in order to be considered primary. For example, the precondition says that $v.set$ must intersect the membership set of all previously-created smaller-id views w for which there is no intervening totally registered view – that is, the set of all "possible previous primary views". Since (for convenience) we allow out of order view creation in DVS, we also include a symmetric condition for previously-created larger-id views.

DVS informs its clients of view changes using DVS-NEWVIEW actions. Even though views can be created out of view id order, the notification to each client is consistent with that order. Not every client needs to see every view. DVS allows the client at each processor p to "register" the current view at p with an action DVS-REGISTER$_p$. With this action, the client at p informs the service that it has obtained whatever information the application needs to begin operating in the new view. For many applications, this will mean that p has received messages from every other member, reporting its state at the start of the new view.

DVS allows a processor p to broadcast a message m using a DVS-GPSND$(m)_p$ action, and delivers the message to a processor q using a DVS-GPRCV$(m)_{p,q}$ action. DVS also uses a DVS-SAFE$(m)_{p,q}$ action to report to processor q that the earlier message m from p has been delivered to all members of the current view of q. DVS guarantees that messages sent by a processor p when the current view of p is v are delivered only within view v (i.e., only to processors in $v.set$ whose current view is v). Moreover, each processor receives messages in the same order as any other processor and without gaps in the sequence of received messages; however, some processors may receive only a prefix of the sequence of messages received by other processors.

Invariant 4.1 expresses the key intersection property guaranteed by DVS; this is weaker than the intersection property required by static definitions of primary views, which says

Signature:

Input: DVS-GPSND$(m)_p$, $m \in \mathcal{M}_c$, $p \in \mathcal{P}$
DVS-REGISTER$_p$, $p \in \mathcal{P}$
Internal: DVS-CREATEVIEW(v), $v \in \mathcal{V}$
DVS-ORDER(m,p,g), $m \in \mathcal{M}_c$, $p \in \mathcal{P}$, $g \in \mathcal{G}$
Output: DVS-GPRCV$(m)_{p,q}$, $m \in \mathcal{M}_c$, $p,q \in \mathcal{P}$
DVS-SAFE$(m)_{p,q}$, $m \in \mathcal{M}_c$, $p,q \in \mathcal{P}$
DVS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$

State:

$created \in 2^{\mathcal{V}}$, init $\{v_0\}$
for each $p \in \mathcal{P}$:
  $current\text{-}viewid[p] \in \mathcal{G}_\perp$, init g_0 if $p \in P_0$, $\perp$ else
for each $g \in \mathcal{G}$:
  $queue[g] \in seqof(\mathcal{M}_c \times \mathcal{P})$, init λ
  $attempted[g] \in 2^{\mathcal{P}}$, init P_0 if $g = g_0$, $\{\}$ else
  $registered[g] \in 2^{\mathcal{P}}$, init P_0 if $g = g_0$, $\{\}$ else
for each $p \in \mathcal{P}$, $g \in \mathcal{G}$:
  $pending[p,g] \in seqof(\mathcal{M}_c)$, init λ
  $next[p,g] \in \mathbf{N}^{>0}$, init 1
  $next\text{-}safe[p,g] \in \mathbf{N}^{>0}$, init 1
Derived variables:
  $Att \in 2^{\mathcal{V}}$, defined as $\{v \in created \mid attempted[v.id] \neq \{\}\}$
  $TotAtt \in 2^{\mathcal{V}}$, defined as $\{v \in created \mid v.set \subseteq attempted[v.id]\}$
  $Reg \in 2^{\mathcal{V}}$, defined as $\{v \in created \mid registered[v.id] \neq \{\}\}$
  $TotReg \in 2^{\mathcal{V}}$, defined as $\{v \in created \mid v.set \subseteq registered[v.id]\}$

Transitions:

internal DVS-CREATEVIEW(v)
  Pre: $\forall w \in created : v.id \neq w.id$
    $\forall w \in created :$
      $\exists x \in TotReg : w.id < x.id < v.id$
      or $\exists x \in TotReg : v.id < x.id < w.id$
      or $v.set \cap w.set \neq \{\}$
  Eff: $created := created \cup \{v\}$

output DVS-NEWVIEW$(v)_p$
  Pre: $v \in created$
    $v.id > current\text{-}viewid[p]$
  Eff: $current\text{-}viewid[p] := v.id$
    $attempted[g] := attempted[g] \cup \{p\}$

input DVS-REGISTER$_p$
  Eff: if $current\text{-}viewid[p] \neq \perp$ then
    $registered[current\text{-}viewid[p]] :=$
      $registered[current\text{-}viewid[p]] \cup \{p\}$

input DVS-GPSND$(m)_p$
  Eff: if $current\text{-}viewid[p] \neq \perp$ then
    append m to $pending[p, current\text{-}viewid[p]]$

internal DVS-ORDER(m,p,g)
  Pre: m is head of $pending[p,g]$
  Eff: remove head of $pending[p,g]$
    append $\langle m,p \rangle$ to $queue[g]$

output DVS-GPRCV$(m)_{p,q}$, choose g
  Pre: $g = current\text{-}viewid[q]$
    $queue[g](next[q,g]) = \langle m,p \rangle$
  Eff: $next[q,g] := next[q,g] + 1$

output DVS-SAFE$(m)_{p,q}$, choose g, P
  Pre: $g = current\text{-}viewid[q]$
    $\langle g,P \rangle \in created$
    $queue[g](next\text{-}safe[q,g]) = \langle m,p \rangle$
    for all $r \in P$:
      $next[r,g] > next\text{-}safe[q,g]$
  Eff: $next\text{-}safe[q,g] := next\text{-}safe[q,g] + 1$

Figure 2: DVS

that all primary components must intersect. This invariant is our version of the correctness requirement for dynamic view services that two consecutive primary views intersect.

Invariant 4.1 (DVS)
If $v,w \in created$, $v.id < w.id$, and there is no $x \in TotReg$ such that $v.id < x.id < w.id$, then $v.set \cap w.set \neq \{\}$.

Invariant 4.2 says that if a view w is totally attempted, then no earlier view v can still be "active".

Invariant 4.2 (DVS)
If $v \in created$, $w \in TotAtt$, and $v.id < w.id$, then there exists $p \in v.set$ with $current\text{-}viewid[p] > v.id$.

5 An implementation of DVS

We now give an algorithm that implements the DVS service, in the sense of inclusion of sets of traces. We build the algorithm on top of the VS service and we use ideas from [18]. The overall system consists of an automaton VS-TO-DVS$_p$ for each $p \in \mathcal{P}$, and VS.

5.1 The implementation

The automaton VS-TO-DVS$_p$ is given in Figure 3. VS-TO-DVS$_p$ uses special non-client messages, tagged either with *"info"* or *"registered"*. Thus, we use $\mathcal{M} = \mathcal{M}_c \cup (\{\text{"info"}\} \times \mathcal{V} \times 2^{\mathcal{V}}) \cup \{\text{"registered"}\}$, where $\mathcal{M}_c$ is the set of all client messages and $\mathcal{M}$ is the universe of all messages. The *attempted*, *reg*, and *info-sent* state variables are not needed for the algorithm, but only for the proofs.

VS-TO-DVS$_p$ acts as a "filter", receiving VS-NEWVIEW inputs from the underlying VS service and deciding whether to accept the proposed views as primary views. If VS-TO-DVS$_p$ decides to accept some such view v, it "attempts" the view by performing a DVS-NEWVIEW(v) output. For each v, we think of the DVS internal DVS-CREATEVIEW(v) action as occurring at the time of the first DVS-NEWVIEW(v) event.

According to the DVS specification, the algorithm is supposed to guarantee nonempty intersection of each newly-created primary view v with any previously-created view w having no intervening totally registered view – a *global* condition involving *nonempty* intersection. The VS-TO-DVS$_p$ processors, however, do not have accurate knowledge of which primary views have been created by other processors, nor of which views are totally registered. Therefore, the processors employ a *local* check of *majority* intersection with known views, rather than a global check of nonempty intersection with existing views. Specifically, each VS-TO-DVS$_p$ keeps track of an "active" view *act*, which is the latest view that it knows to be totally registered, plus a set of "ambiguous" views *amb*, which are all the views that it knows have been attempted (i.e., have had a DVS-NEWVIEW action performed someplace), and whose ids are greater than $act.id$. We define $use = \{act\} \cup amb$. When VS-TO-DVS$_p$ receives a VS-NEWVIEW(v) input, it sends out *"info"* messages containing its current *act* and *amb* values to all the other processors in the new view, using the VS service, and then waits to receive corresponding *"info"* messages for view v from all the other processors in the view. After receiving this information

Signature:

Input: DVS-GPSND$(m)_p$, $m \in \mathcal{M}_c$
DVS-REGISTER$_p$
VS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$
VS-GPRCV$(m)_{q,p}$, $m \in \mathcal{M}$, $q \in \mathcal{P}$
VS-SAFE$(m)_{q,p}$, $m \in \mathcal{M}$, $q \in \mathcal{P}$

Internal: DVS-GARBAGE-COLLECT$(v)_p$, $v \in \mathcal{V}$
Output: VS-GPSND$(m)_p$, $m \in \mathcal{M}$
DVS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$
DVS-GPRCV$(m)_{q,p}$, $m \in \mathcal{M}_c$, $q \in \mathcal{P}$
DVS-SAFE$(m)_{q,p}$, $m \in \mathcal{M}_c$, $q \in \mathcal{P}$

State:

$cur \in \mathcal{V}_\perp$, init v_0 if $p \in P_0$, $\perp$ else
client-cur $\in \mathcal{V}_\perp$, init v_0 if $p \in P_0$, $\perp$ else
$act \in \mathcal{V}$, init v_0
$amb \in 2^{\mathcal{V}}$, init $\{\}$
attempted $\in 2^{\mathcal{V}}$, init $\{v_0\}$ if $p \in P_0$, $\{\}$ else
for each $g \in \mathcal{G}$
msgs-to-vs$[g] \in seqof(\mathcal{M})$, init λ
msgs-from-vs$[g] \in seqof(\mathcal{M}_c \times P)$, init λ
safe-from-vs$[g] \in seqof(\mathcal{M}_c \times P)$, init λ
reg$[g]$ a bool, init **true** if $p \in P_0$ and $g = g_0$, **false** else
info-sent$[g] \in (\mathcal{V} \times 2^{\mathcal{V}})_\perp$, init $\perp$
for each $g \in \mathcal{G}$, $q \in \mathcal{P}$
info-rcvd$[q,g] \in (\mathcal{V} \times 2^{\mathcal{V}})_\perp$, init $\perp$
rcvd-rgst$[q,g]$ a bool, init **false**

Derived variables
$use \in 2^{\mathcal{V}}$, defined as $use = \{act\} \cup amb$

Transitions:

input VS-NEWVIEW$(v)_p$
Eff: $cur := v$
append $\langle$ *"info"*, *act*, *amb*$\rangle$ to *msgs-to-vs*$[cur.id]$
info-sent$[cur.id] := \langle act, amb \rangle$

input VS-GPRCV$(\langle$ *"info"*$, v, V\rangle)_{q,p}$
Eff: *info-rcvd*$[q, cur.id] := \langle v, V \rangle$
if $v.id > act.id$ then $act := v$
$amb := \{w \in amb \cup V \mid w.id > act.id\}$

input VS-SAFE$(\langle$ *"info"*$, v, V\rangle)_{q,p}$
Eff: none

output DVS-NEWVIEW$(v)_p$
Pre: $v = cur$
$v.id > client\text{-}cur.id$
$\forall q \in v.set, q \neq p$: *info-rcvd*$[q, v.id] \neq \perp$
$\forall w \in use$: $|v.set \cap w.set| > |w.set|/2$
Eff: $amb := amb \cup \{v\}$
$attempted := attempted \cup \{v\}$
$client\text{-}cur := v$

input DVS-REGISTER$_p$
Eff: if *client-cur* $\neq \perp$ then
reg$[client\text{-}cur] :=$ **true**
append $\langle$ *"registered"*$\rangle$ to *msgs-to-vs*$[client\text{-}cur.id]$

input VS-GPRCV$(\langle$ *"registered"*$\rangle)_{q,p}$
Eff: *rcvd-rgst*$[cur.id, q] :=$ **true**

input VS-SAFE$(\langle$ *"registered"*$\rangle)_{q,p}$
Eff: none

internal DVS-GARBAGE-COLLECT$(v)_p$
Pre: $\forall q \in v.set$: *rcvd-rgst*$[v, q] =$ **true**
$v.id > act.id$
Eff: $act := v$
$amb := \{w \in amb \mid w.id > act.id\}$

input DVS-GPSND$(m)_p$
Eff: if *client-cur*$.id_p \neq \perp$ then
append m to *msgs-to-vs*$[client\text{-}cur.id]$

output VS-GPSND$(m)_p$
Pre: m is head of *msgs-to-vs*$[cur.id]$
Eff: remove head of *msgs-to-vs*$[cur.id]$

input VS-GPRCV$(m)_{q,p}$, where $m \in \mathcal{M}_c$
Eff: append $\langle m, q \rangle$ to *msgs-from-vs*$[cur.id]$

output DVS-GPRCV$(m)_{q,p}$
Pre: $\langle m, q \rangle$ is head of *msgs-from-vs*$[client\text{-}cur.id]$
Eff: remove head of *msgs-from-vs*$[client\text{-}cur.id]$

input VS-SAFE$(m)_{q,p}$, where $m \in \mathcal{M}_c$
Eff: append $\langle m, q \rangle$ to *safe-from-vs*$[cur.id]$

output DVS-SAFE$(m)_p$
Pre: $\langle m, q \rangle$ is head of *safe-from-vs*$[client\text{-}cur.id]$
Eff: remove head of *safe-from-vs*$[client\text{-}cur.id]$

Figure 3: VS-TO-DVS$_p$

(and updating its own *act* and *amb* accordingly), VS-TO-DVS$_p$ checks that v has a majority intersection with each view in *use*. If so, VS-TO-DVS$_p$ performs a DVS-NEWVIEW$_p$ output.

Then the clients can use the communication system to exchange state information as needed for processing in view v. When client at p has obtained enough information, it "registers" the view by means of action DVS-REGISTER$_p$, which causes processor p to send *"registered"* messages to the other members. When a processor receives *"registered"* messages for a view v from all members, it may perform garbage collection by discarding information about views with ids smaller than that of v. VS-TO-DVS uses VS to send and receive messages.

We define the system DVS-IMPL to be the composition of all the VS-TO-DVS$_p$ automata and VS with all the external actions of VS hidden. We introduce four derived variables for DVS-IMPL analogous to those of DVS, indicating the attempted, totally attempted, registered, and totally registered views, respectively. They are:
$\mathcal{A}tt = \{v \in created \mid (\exists p \in v.set) v \in attempted_p\}$;
$\mathcal{T}ot\mathcal{A}tt = \{v \in created \mid (\forall p \in v.set) v \in attempted_p\}$;
$\mathcal{R}eg = \{v \in created \mid (\exists p \in v.set) reg[v.id]_p =$ **true**$\}$; and
$\mathcal{T}ot\mathcal{R}eg = \{v \in created \mid (\forall p \in v.set) reg[v.id]_p =$ **true**$\}$.

5.2 Invariants

This section contains the main invariants of DVS-IMPL needed for the refinement proof in Section 5.3.

Invariant 5.1 (DVS-IMPL)
If $v \in attempted_p$ *and* $q \in v.set$ *then* $cur.id_q \geq v.id$.

Invariant 5.2 (DVS-IMPL)
1. $act_p \in \mathcal{T}ot\mathcal{R}eg$.
2. *If* $w \in amb_p$ *then* $act.id_p < w.id$.
3. *If* $client\text{-}cur_p \neq \perp$ *and* $w \in \{act_p\} \cup amb_p$, *then* $w.id \leq client\text{-}cur.id_p$.
4. *If* *info-sent*$[g]_p = \langle x, X \rangle$ *then* $x \in \mathcal{T}ot\mathcal{R}eg$.
5. *If* *info-sent*$[g]_p = \langle x, X \rangle$ *and* $w \in X$ *then* $x.id < w.id$.
6. *If* *info-sent*$[g]_p = \langle x, X \rangle$ *and* $w \in \{x\} \cup X$ *then* $w.id < g$.

Invariant 5.3 says that certain views appear in *"info"* messages, unless they have been garbage-collected.

Invariant 5.3 (DVS-IMPL)
1. If info-sent$[g]_p = \langle x, X\rangle$ *and* $w \in attempted_p$, *then either* $w \in \{x\} \cup X$ *or* $w.id < x.id$.
2. If info-rcvd$[q, g]_p = \langle x, X\rangle$ *and* $w \in \{x\} \cup X$, *then either* $w \in use_p$ *or* $w.id < act.id_p$.

Invariant 5.4 says that two attempted views having no intervening totally registered view, and having a common member, q, that has attempted the first view, must intersect in a majority of processors. This is because, under these circumstances, information must flow from q to any process that attempts the second view.

Invariant 5.4 (DVS-IMPL)
Suppose that $v \in attempted_p$, $q \in v.set$, $w \in attempted_q$, $w.id < v.id$, *and there is no* $x \in \mathcal{TotReg}$ *such that* $w.id < x.id < v.id$. *Then* $|v.set \cap w.set| > |w.set|/2$.

Proof: By induction on the length of an execution.
Base: In the initial state, only v_0 is attempted, so the hypotheses cannot be satisfied. Thus, the statement is vacuously true.
Inductive step: Fix any step (s, π, s'), where s is reachable, and assume the invariant is true in state s. We show that it is true in s'. So fix v, w, p, and q, and assume that $v \in s'.attempted_p$, $q \in v.set$, $w \in s'.attempted_q$, $w.id < v.id$, and there is no $x \in s'.\mathcal{TotReg}$ such that $w.id < x.id < v.id$. Then also there is no $x \in s.\mathcal{TotReg}$ such that $w.id < x.id < v.id$. We consider four cases:
1. $v \in s.attempted_p$ and $w \in s.attempted_q$. Then the statement for s implies that $|v.set \cap w.set| > |w.set|/2$, as needed.
2. $v \notin s.attempted_p$ and $w \notin s.attempted_q$. This cannot happen because we cannot have both v and w becoming attempted in a single step.
3. $v \notin s.attempted_p$ and $w \in s.attempted_q$. Then π must be DVS-NEWVIEW$(v)_p$. By the precondition of π we have that $s.info\text{-}rcvd[q, v.id]_p = \langle x, X\rangle$ for some x and X. It follows that $s.info\text{-}sent[v.id]_q = \langle x, X\rangle$. Then Invariant 5.3 implies that either $w \in \{x\} \cup X$ or $w.id < x.id$. If $w.id < x.id$, then we obtain a contradiction, because $x \in s.\mathcal{TotReg}$ (by Invariant 5.2) and $x.id < v.id$ (by Invariant 5.2). So $w \in \{x\} \cup X$. Now by Invariant 5.3 we have that either $w \in s.use_p$ or $w.id < s.act.id_p$. In the former case, by the precondition of π, we have $|v.set \cap w.set| > |w.set|/2$. In the latter case, we obtain a contradiction, because $s.act_p \in \mathcal{TotReg}$ (by Invariant 5.2) and Invariant 5.2 implies that $s.act.id_p \leq s.client\text{-}cur_p < s.cur_p = v.id$. 4. $v \in s.attempted_p$ and $w \notin s.attempted_q$. Then π must be DVS-NEWVIEW$(w)_q$. But this cannot happen: Since $v \in s.attempted_p$ and $q \in v.set$, Invariant 5.1 implies that $s.cur.id_q \geq v.id$. Since $v.id > w.id$, we have $s.cur.id_q > w.id$. But the precondition of action π requires $s.cur.id_q = w.id$, so π is not enabled in s. □

Invariant 5.5 says that any attempted view v intersects the latest preceding totally registered view w in a majority of members of w.

Invariant 5.5 (DVS-IMPL)
Suppose that $v \in \mathcal{Att}$, *and* $w \in \mathcal{TotReg}$, $w.id < v.id$, *and there is no* $x \in \mathcal{TotReg}$ *such that* $w.id < x.id < v.id$. *Then* $|v.set \cap w.set| > |w.set|/2$.

Proof: By induction on the length of an execution.
Base: In the initial state, fix v and w satsifying the hypotheses. The first two assumptions imply that $v = w = v_0$. But then the third assumption $w.id < v.id$ is false. Thus, the hypothesis of the statement is false, so the statement is vacuously true.
Inductive step: Fix any step (s, π, s'), where s is reachable, assume the invariant is true in state s, and show that it is true in s'. So fix v and w, and assume that $v \in s'.\mathcal{Att}$, $w \in s'.\mathcal{TotReg}$, $w.id < v.id$, and there is no $x \in s'.\mathcal{TotReg}, w.id < x.id < v.id$. We consider four cases:
1. $v \in s.\mathcal{Att}$ and $w \in s.\mathcal{TotReg}$. Then the statement for s implies that $|v.set \cap w.set| > |w.set|/2$, as needed.
2. $v \notin s.\mathcal{Att}$ and $w \notin s.\mathcal{TotReg}$. This cannot happen because we cannot have both v becoming attempted and w becoming totally registered in a single step.
3. $v \notin s.\mathcal{Att}$ and $w \in s.\mathcal{TotReg}$. Then π must be DVS-NEWVIEW$(v)_p$ for some p. The precondition of π implies that, for any view $y \in s.use_p$, $|v.set \cap y.set| > |y.set|/2$. Hence to prove the claim it is enough to prove that $w \in s.use_p$. We proceed by contradiction assuming that $w \notin s.use_p$.
By Invariant 5.2, $s.use_p \cap s.\mathcal{TotReg} \neq \{\}$. Let m be the view in $s.\mathcal{TotReg} \cap s.use_p$ having the biggest identifier. We know that $m \neq w$ because $w \notin s.use_p$. It follows then that $m.id \neq w.id$. We claim that $m.id < w.id$. Suppose for the sake of contradiction that $m.id > w.id$. Since $m \in s.use_p$, Invariant 5.2 implies that $m.id \leq s.client\text{-}cur_p < s.cur_p = v.id$. So $w.id < m.id < v.id$. Since $m \in s'.\mathcal{TotReg}$, this contradicts the hypothesis of the inductive step. Therefore, $m.id < w.id$. Let n be the view in $s.\mathcal{TotReg}$ that has the smallest id strictly greater than that of m. Note that $m.id < n.id \leq w.id < v.id$. Since $m \in s.use_p$, the precondition of π implies that $|v.set \cap m.set| > |m.set|/2$. By the statement applied to state s, $|n.set \cap m.set| > |m.set|/2$. Hence there exists a processor $q \in v.set \cap n.set$. By the precondition of π, $s.info\text{-}rcvd[q, v.id]_p = \langle x, X\rangle$ for some x, X. Invariant 5.2 implies that $x.id < v.id$. Since $n \in s.\mathcal{TotReg}$, we have that $n \in s.attempted_q$. Then Invariant 5.3 (used with $w = n$) implies that either $w \in \{x\} \cup X$ or $w.id < x.id$. In either case, $\{x\} \cup X$ contains a view $y \in s.\mathcal{TotReg}$ (either w or x) such that $n.id \leq y.id < v.id$. Then Invariant 5.3 implies that either $y \in s.use_p$ or $y.id < s.act.id_p$. By Invariant 5.2, $s.act_p \in s.\mathcal{TotReg}$ and by definition, $s.act_p \in s.use_p$. So in either case, the hypothesis that m is the totally registered view with the largest id belonging to $s.use_p$ is contradicted.
4. $v \in s.\mathcal{Att}$ and $w \notin s.\mathcal{TotReg}$. Then π must be DVS-REGISTER$_p$ for some p. Let m be the view in $s.\mathcal{TotReg}$ with the largest id that is strictly less than $w.id$. By the statement for s, we know that $|w.set \cap m.set| > |m.set|/2$ and $|v.set \cap m.set| > |m.set|/2$. Hence there is a processor $q \in w.set \cap v.set$.
Since $v \in s.\mathcal{Att}$, there exists a processor r such that $v \in s.attempted_r$; thus also $v \in s'.attempted_r$. Since $w \in s'.\mathcal{TotReg}$, we have that $w \in s'.attempted_q$. By assumption, there is no view $x \in s'.\mathcal{TotReg}$ such that $w.id < x.id < v.id$. By Invariant 5.4 applied to state s' (with $p = r$), we have that $|v.set \cap w.set| > |w.set|/2$, as needed. □

The final invariant, a corollary to Invariant 5.5, is instrumental in the refinement proof.

Invariant 5.6 (DVS-IMPL)
If $v, w \in \mathcal{Att}$, $w.id < v.id$, *and there is no* $x \in \mathcal{TotReg}$ *with* $w.id < x.id < v.id$, *then* $v.set \cap w.set \neq \{\}$.

Proof: Suppose that v and w are as given. We consider two cases.
1. $w \in \mathcal{TotReg}$. Then since there is no $x \in \mathcal{TotReg}$ with $w.id < x.id < v.id$, it follows that w is the view with the largest id in the set $\{y \in \mathcal{TotReg} : y.id < v.id\}$. Then Invariant 5.5 implies that $|v.set \cap w.set| > |w.set|/2$, which implies that $v.set \cap w.set \neq \{\}$, as needed.
2. $w \notin \mathcal{TotReg}$. Then let $Y = \{y \in \mathcal{TotReg} : y.id < w.id\}$. We claim that Y is nonempty: We know that $v_0 \in \mathcal{TotReg}$ and that $v_0.id \leq w.id$. If $v_0.id = w.id$, then by Lemma 3.1, we

have $w = v_0$. But then $w \in \mathcal{TotReg}$, a contradiction. So we must have $v_0.id < w.id$, which implies that $v_0 \in Y$, so Y is nonempty.

Now fix z to be the view in Y with the largest id. Since there is no $x \in \mathcal{TotReg}$ with $w.id < x.id < v.id$, it follows that z is also the view with the largest id in the set $\{y \in \mathcal{TotReg} : y.id < v.id\}$. Then Invariant 5.5 implies that $|w.set \cap z.set| > |z.set|/2$ and $|v.set \cap z.set| > |z.set|/2$. Together, these two facts imply that $v.set \cap w.set \neq \{\}$, as needed. □

5.3 The refinement

We prove that DVS-IMPL implements DVS by defining a function $\mathcal{F}$ that maps states of DVS-IMPL to states of DVS and proving that this function is a *refinement*.

DVS-IMPL uses VS to send client messages and messages generated by the implementation (*"info"* and *"registered"* messages). The refinement discards the non-client messages. Thus, if q is a finite sequence of client and non-client messages, we define $purge(q)$ to be the queue obtained by deleting any *"info"* or *"registered"* messages from q, and $purgesize(q)$ to be the number of *"info"* and *"registered"* messages in q.

Figure 4 defines the refinement $\mathcal{F}$. The fact that $\mathcal{F}$ is a refinement is shown using two lemmas, 5.7 and 5.8, expressing the two conditions required by the definition of a refinement:

Lemma 5.7 *If s is an initial state of* DVS-IMPL *then $\mathcal{F}(s)$ is an initial state of* DVS.

Lemma 5.8 *Let s be a reachable state of* DVS-IMPL, *$\mathcal{F}(s)$ a reachable state of* DVS, *and (s, π, s') a step of* DVS-IMPL. *Then there is an execution fragment α of* DVS *that goes from $\mathcal{F}(s)$ to $\mathcal{F}(s')$, such that $trace(\alpha) = trace(\pi)$.*

Proof: By case analysis based on the type of the action π. The only interesting case is where $\pi = \text{DVS-NEWVIEW}(v)_p$. Then $trace((s, \pi, s)) = \pi$. Define $t = \mathcal{F}(s)$ and $t' = \mathcal{F}(s')$. We consider two cases:

1. $v \in t.created$. In this case, we set $\alpha = (t, \pi', t')$, where $\pi' = \text{DVS-NEWVIEW}(v)_p$. The code shows that π' brings DVS from state t to state t'. It remains to prove that π' is enabled in state t, that is, that $v \in t.created$ and $v.id > t.current\text{-}viewid[p]$. The first of these two conditions is true because of the defining condition for this case. The second condition follows from the precondition of π in DVS-IMPL: this precondition implies that $v.id > s.client\text{-}cur.id_p$, and by the definition of $\mathcal{F}$ we have $t.current\text{-}viewid[p] = s.client\text{-}cur.id_p$.
2. $v \notin t.created$. In this case we set $\alpha = (t, \pi', t'', \pi'', t')$, where $\pi' = \text{DVS-CREATEVIEW}(v)_p$, $\pi'' = \text{DVS-NEWVIEW}(v)_p$, and t'' is the unique state that arises by running the effect of π' from t. The code shows that α brings DVS from state t to state t'. It remains to prove that π' is enabled in t and that π'' is enabled in t''.

The precondition of π' requires that (i) $\forall w \in t.created$, $v.id \neq w.id$ and (ii) $\forall w \in t.created$, either $\exists x \in s.\mathcal{TotReg}$ satisfying $w.id < x.id < v.id$ or $v.id < x.id < w.id$, or else $v.set \cap w.set \neq \{\}$. To see requirement (i), suppose for the sake of contradiction that $w \in t.created$ and $w.id = v.id$. The precondition of π in DVS-IMPL implies that $v = s.cur_p$, which implies that $v \in s.created$. Since $w \in t.created$, the definition of $\mathcal{F}$ implies that $w \in s.attempted_q$ for some q. This implies that $w \in s.created$. But then Invariant 3.1 implies that $v = w$. But this contradicts that fact that $v \notin t.created$ and $w \in t.created$. To see requirement (ii), suppose that $w \in t.created$ and there is no $x \in s.\mathcal{TotReg}$ satisfying $w.id < x.id < v.id$ or $v.id < x.id < w.id$. Then $w \in s.attempted_q$ for some q, by definition of $\mathcal{F}$. Part (i) implies that $w \neq v$, so also $w \in s'.attempted_q$. Therefore, $w \in s'.\mathcal{A}tt$. We also have $v \in s'.\mathcal{A}tt$. Moreover, there is no $x \in s'.\mathcal{TotReg}$ satisfying $w.id < x.id < v.id$ or $v.id < x.id < w.id$. Then Invariant 5.6 implies that $v.set \cap w.set \neq \{\}$, as needed to prove that π' is enabled in t.

We now prove that π'' is enabled in state t''. The precondition of π'' requires that $v \in t''.created$ and $v.id > t''.current\text{-}viewid[p]$. The first condition is true because v is added to *created* by π'. The second condition follows from the precondition of π in DVS-IMPL: The precondition of π implies that $v.id > s.client\text{-}cur.id_p$. The definition of $\mathcal{F}$ implies that $t.current\text{-}viewid[p] = s.client\text{-}cur.id_p$. Moreover, $t''.current\text{-}viewid[p] = t.current\text{-}viewid[p]$. It follows that $v.id > t''.current\text{-}viewid[p]$. Thus π'' is enabled in state t''. □

Theorem 5.9 *Every trace of* DVS-IMPL *is a trace of* DVS.

Proof: Lemmas 5.7 and 5.8 imply that $\mathcal{F}$ is a refinement, which implies the result. □

6 An application of DVS

Now we show how to use DVS to implement the totally ordered broadcast service TO, defined in [12]. This service accepts messages from clients and delivers them to all clients according to the same total order. The implementation consists of an automaton DVS-TO-TO$_p$ for each $p \in \mathcal{P}$, and the DVS specification.

6.1 The implementation

The implementation is very similar to the TO implementation provided in [12]; we refer the reader to the informal algorithm description in that paper. Both algorithms rely on primary views to establish a total order of client messages. The main difference is that the algorithm in [12] uses a static notion of primary and the new one uses a dynamic notion. The algorithm of [12] is built upon a VS service that reports non-primary as well as primary views, and uses a simple local test to determine if the view is primary. That algorithm does some non-critical background work (gossiping information) in non-primary views. In contrast, the new algorithm is built upon our DVS service, which only reports primary views. Thus the new algorithm is simpler in that it does not perform the local tests and does not carry out any processing in non-primary views. On the other hand, in the new algorithm, the application programs must perform DVS-REGISTER actions to tell the DVS service when they have "established" new views. Although the new algorithm appears very similar to the old one, the fact that the DVS service provides weaker and more complicated guarantees than the VS service makes the new algorithm harder to prove correct. Another, minor difference between this algorithm and that of [12] is the new handling of initial views. In particular, a *delay* buffer is used for client messages, to accommodate messages that might arrive before a node has any view.

The code for automaton DVS-TO-TO$_p$ appears in Figure 5. In this code, $\mathcal{L} = \mathcal{G} \times \mathbf{N}^{>0} \times \mathcal{P}$ is the set of *labels*, with selectors $l.id$, $l.seqno$ and $l.origin$. $\mathcal{A}$ is the set of messages that can be sent by the clients of the TO service. $\mathcal{C} = \mathcal{L} \times \mathcal{A}$ is the set of possible associations between labels and client messages. $\mathcal{S} = 2^{\mathcal{C}} \times seqof(\mathcal{L}) \times \mathbf{N}^{>0} \times \mathcal{G}$ is the set of *summaries*, with selectors $x.con$, $x.ord$, $x.next$ and $x.high$.

If Y is a partial function from processor ids to summaries, then we define:

$knowncontent(Y) = \cup_{q \in dom(Y)} Y(q).con$,
$maxprimary(Y) = \max_{q \in dom(Y)}\{Y(q).high\}$,
$maxnextconfirm(Y) = \max_{q \in dom(Y)} Y(q).next$,

Let s be a state of DVS-IMPL. The state $t = \mathcal{F}(s)$ of DVS is the following.

- $t.created = \cup_{p \in \mathcal{P}} s.attempted_p$
- for each $p \in \mathcal{P}$, $t.current\text{-}viewid[p] = s.client\text{-}cur.id_p$
- for each $g \in \mathcal{G}$, $t.registered[g] = \{p | s.reg[g]_p\}$
- for each $p \in \mathcal{P}$, $g \in \mathcal{G}$, $t.pending[p,g] = purge(s.pending[p,g]) + purge(s.msgs\text{-}to\text{-}vs[g]_p)$
- for each $g \in \mathcal{G}$, $t.queue[g] = purge(s.queue[g])$
- for each $p \in \mathcal{P}$, $g \in \mathcal{G}$, $t.next[p,g] = s.next[p,g] - purgesize(s.queue[g](1..next[p,g]-1)) - |s.msgs\text{-}from\text{-}vs[g]_p|$
- for each $p \in \mathcal{P}$, $g \in \mathcal{G}$, $t.next\text{-}safe[p,g] = s.next\text{-}safe[p,g] - purgesize(s.queue[g](1..next\text{-}safe[p,g]-1)) - |s.safe\text{-}from\text{-}vs[g]_p|$

Figure 4: The refinement $\mathcal{F}$.

$reps(Y) = \{q \in dom(Y) : Y(q).high = maxprimary\}$,
$chosenrep(Y)$ = some element in $reps(Y)$,
$shortorder(Y) = Y(chosenrep(Y)).ord$, and
$fullorder(Y) = shortorder(Y)$ followed by the remaining elements of $dom(knowncontent(Y))$, in label order.

The algorithm involves *normal* and *recovery* activity. Normal activity occurs while a group view is not changing. Recovery activity begins when a new primary view is presented by DVS, and continues while the members combine information from their previous history, to provide a consistent basis for ongoing normal activity.

During normal activity, each client message received by TO-IMPL is given a system-wide unique *label*, which is remembered in a relation *content* and communicated to other processes in the same view using DVS. When a message is received, the label is given an *order*, a tentative position in the system-wide total order the service is to provide. The consistent sequence of message delivery within each view keeps this tentative order consistent at members of a given view, but it need not always be consistent between nodes in different views. When all members of a view have given a label an order, the label and its order may become *confirmed*. This is deduced by the node from the occurrence of the safe indication for the message that carried the label. The messages associated with confirmed labels may be released to the clients in the given order.

When a new primary view is reported by DVS, recovery activity occurs to integrate the knowledge of different members. First, each member of a new view sends a message, using DVS, that contains a summary of that node's state, including the tentative order built in its previous view. Once a node has received all members' state summaries, it processes the information in one atomic step, i.e., it *establishes* the new view. Once a node establishes a view, it informs DVS of that fact with a DVS-REGISTER action. Then recovery continues by collecting the DVS safe indications. Once the state exchange is safe, all labels used in the exchange are marked as safe and all associated messages are confirmed just as in normal processing.

The system TO-IMPL is the composition of all the DVS-TO-TO$_p$ automata and DVS with all the external actions of DVS hidden. The *allstate* and *allconfirm* derived variables are defined for TO-IMPL just as in [12].

6.2 Correctness proof

The correctness proof for TO-IMPL follows the outline of the one in [13]. The main difference is that the main invariant, which corresponds to Lemma 6.17 of [12], requires a different, more subtle proof. Invariant 6.1 says that any view that is known (anywhere in the system state) to be an established primary was in fact attempted by all its members.

Invariant 6.1 (TO-IMPL)
If $x \in allstate$ then there exists $w \in created$ such that $x.high = w.id$, and for all $p \in w.set$, $p \in attempted[w.id]$.

Invariant 6.2 says that if a view w is established, then no earlier view v can still be active.

Invariant 6.2 (TO-IMPL)
If $v \in created$, $x \in allstate$ and $x.high > v.id$ then there exists $p \in v.set$ with $current.id_p > v.id$.

Proof: Fix v, x as given. Lemma 6.1 shows the existence of $w \in created$ such that $x.high = w.id$, and for all $p \in w.set$, $p \in attempted[w.id]$. Then Invariant 4.2 implies that there exists $p \in v.set$ with $current\text{-}viewid[p] > v.id$. But $current\text{-}viewid[p] = current.id_p$, which yields the result. □

Finally we provide the proof for the invariant corresponding to the invariant stated in Lemma 6.17 of [13].

Invariant 6.3 (TO-IMPL)
Suppose that $v \in created$, $\sigma \in seqof(\mathcal{L})$, and for every $p \in v.set$, the following is true:
If $current.id_p > v.id$ then $established[v.id]_p$
and $\sigma \leq buildorder[p, v.id]$.
Then for every $x \in allstate$ with $x.high > v.id$, $\sigma \leq x.ord$.

Proof: By induction on the length of the execution. We present the differences from the proof in [13].
Base: In the initial state, the only created view is v_0, and there is no $x \in allstate$ with $x.high > g_0$. So the statement is vacuously true.
Inductive step: Fix any step (s, π, s'), where s is reachable, assume the invariant is true in state s, and show that it is true in s'. So fix $v \in s'.created$ and σ, and assume that for every $p \in v.set$, if $s'.current.id_p > v.id$ then $s'.established[v.id]_p$ and $\sigma \leq s'.buildorder[p, v.id]$.

If $v \notin s.created$, then π must be CREATEVIEW(v). Then $s'.established[v.id]_p = false$ for all p. Fix $x \in s'.allstate$ and suppose that $x.high > v.id$. Then Invariant 6.2 applied to s' implies that there exists $p \in v.set$ with $s'.current.id_p > v.id$; fix such a p. Then the hypothesis part of the invariant for s' implies that $s'.established[v.id]_p = true$, a contradiction. It follows that $v \in s.created$.

As before, the interesting steps are GPRCV$_p$ steps that produce a new summary x by delivering the last state-exchange message of a view w to some processor p. Thus $x.high = w.id$. Let x' be the summary of $q' = chosenrep$ in $s'.gotstate$. We claim $x'.high \geq v.id$.

To prove the claim, we let v' denote the unique element with highest viewid among the elements of $s'.created$ such

Signature:

Input: BCAST$(a)_p$, $a \in \mathcal{A}$
DVS-GPRCV$(m)_{q,p}$, $q \in \mathcal{P}$, $m \in \mathcal{C} \cup \mathcal{S}$
DVS-SAFE$(m)_{q,p}$, $q \in \mathcal{P}$, $m \in \mathcal{C} \cup \mathcal{S}$
DVS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$

Output: DVS-REGISTER$_p$
DVS-GPSND$(m)_p$, $m \in \mathcal{C} \cup \mathcal{S}$
BRCV$(a)_{q,p}$, $a \in \mathcal{A}$, $q \in \mathcal{P}$

Internal: CONFIRM$_p$

State:

$current \in \mathcal{V}_\perp$, init v_0 if $p \in P_0$, $\perp$ else
$status \in \{normal, send, collect\}$, init $normal$
$content \in \mathcal{C}$, init $\{\}$
$nextseqno \in \mathbf{N}^{>0}$, init 1
$buffer \in seqof(\mathcal{L})$, init λ
$safe\text{-}labels \in 2^{\mathcal{L}}$, init $\{\}$
$order \in seqof(\mathcal{L})$, init λ
$nextconfirm \in \mathbf{N}^{>0}$, init 1

$nextreport \in \mathbf{N}^{>0}$, init 1
$highprimary \in \mathcal{G}$, init g_0
$gotstate$, a partial function from $\mathcal{P}$ to $\mathcal{S}$, init $\{\}$
$safe\text{-}exch \subseteq \mathcal{P}$, init $\{\}$
$registered \subseteq \mathcal{G}$, init $\{g_0\}$ if $p \in P_0$, $\{\}$ else
$delay \in seqof(\mathcal{A})$, init λ
for each $g \in \mathcal{G}$,
$established[g]$, a bool, init **false**

Transitions:

input BCAST$(a)_p$
Eff: append a to $delay$

internal LABEL$(a)_p$
Pre: a is head of $delay$
$current \neq \perp$
Eff: let m be $\langle current.id, nextseqno, p\rangle$
$content := content \cup \{\langle m, a\rangle\}$
append m to $buffer$
$nextseqno := nextseqno + 1$
delete head of $delay$

output DVS-GPSND$(\langle l, a\rangle)_p$
Pre: $status = normal$
l is head of $buffer$
$\langle l, a\rangle \in content$
Eff: delete head of $buffer$

input DVS-GPRCV$(\langle l, a\rangle)_{q,p}$
Eff: $content := content \cup \{\langle l, a\rangle\}$
$order := order{+}l$

input DVS-SAFE$(\langle l, a\rangle)_{q,p}$
Eff: $safe\text{-}labels := safe\text{-}labels \cup \{l\}$

internal CONFIRM$_p$
Pre: $order(nextconfirm) \in safe\text{-}labels$
Eff: $nextconfirm := nextconfirm + 1$

output BRCV$(a)_{q,p}$
Pre: $nextreport < nextconfirm$
$\langle order(nextreport), a\rangle \in content$
$q = order(nextreport).origin$
Eff: $nextreport := nextreport + 1$

input DVS-NEWVIEW$(v)_p$
Eff: $current := v$
$nextseqno := 1$
$buffer := \lambda$
$gotstate := \{\}$
$safe\text{-}exch := \{\}$
$safe\text{-}labels := \{\}$
$status := send$

output DVS-GPSND$(x)_p$
Pre: $status = send$
$x = \langle content, order, nextconfirm, highprimary\rangle$
Eff: $status := collect$

input DVS-GPRCV$(x)_{q,p}$
Eff: $content := content \cup x.con$
$gotstate := gotstate \oplus \langle q, x\rangle$
if $(dom(gotstate) = current.set) \wedge (status = collect)$ then
$nextconfirm := maxnextconfirm(gotstate)$
$order := fullorder(gotstate)$
$highprimary := current.id$
$status := normal$
$established[current.id] := true$

output DVS-REGISTER$_p$
Pre: $current \neq \perp$
$established[current]$
$current \notin registered$
Eff: $registered := registered \cup \{current\}$

input DVS-SAFE$(x)_{q,p}$
Eff: $safe\text{-}exch := safe\text{-}exch \cup \{q\}$
if $safe\text{-}exch = current.set$ then
$safe\text{-}labels := safe\text{-}labels \cup range(fullorder(gotstate))$

Figure 5: DVS-TO-TO$_p$

that $v'.id < w.id$ and $s'.registered[v'.id] = v'.set$. Let v'' denote either v' or v, whichever has the higher viewid. Invariant 4.1 shows that $w.set \cap v''.set \neq \{\}$, no matter whether v'' is v or v'. Fix any element q'' in $w.set \cap v''.set$. Recall that the condition for establishing a view shows that $domain(s'.gotstate_p) = w.set$, so by the code, either $q'' \in domain(s.gotstate_p)$ or else q'' is the sender of the message whose receipt is the step we are examining. In the former case, let x'' be the summary $s.gotstate(q'')_p$; in the latter let x'' be the summary whose receipt is the event. In either case we have $x'' \in s.allstate[q'', w.id]$.

We now show that $s.established[v''.id]_{q''}$. We consider two cases:

1. $v'' = v'$. Then $q'' \in v'.set$ so by definition of v', we obtain $q'' \in s.registered[v'.id]$; therefore, $s.established[v'.id]_{q''}$.
2. $v'' = v$. Because $s.allstate[q'', w.id]$ is non-empty, the analogue of part 4 of Lemma 6.6 from [13] implies that $s.current.id_{q''} \geq w.id$. We have that $x.high > v.id$ by assumption, and $x.high = w.id$ by the code; therefore, $w.id > v.id$. So also $s.current.id_{q''} > v.id$. Recall that we are in the case where the hypothesis of this lemma is true. Therefore, by this hypothesis (uses $q'' \in v.set$), we obtain that $s.established[v.id]_{q''}$

By the analogue of Lemma 6.13 from [13], (applied with q'' replacing p) we obtain $x''.high \geq v''.id$. By the definition of q' as a member that maximizes the $high$ component in the summary recorded in $s'.gotstate$, we have $x'.high \geq x''.high$. Therefore $x'.high \geq v''.id \geq v.id$, completing our proof of the claim. The rest of the proof is as before. □

To complete the implementation proof, we define a function from the reachable states of TO-IMPL to the states of TO and prove that it is a refinement. This function is defined exactly as in [12], except that, for each p, the abstract pending queue $t.pending[p]$ is defined to include an additional tail consisting of the contents of $s.delay_p$.

Theorem 6.4 *Every trace of* TO-IMPL *is a trace of* TO.

7 Discussion

This work deals entirely with safety properties; it remains to consider performance and fault-tolerance properties as well. It also remains to study other applications of our DVS specification, such as replicated data applications and load-balancing applications.

Another interesting exploration direction considers variations on the DVS specification, for example, one in which the state exchange at the beginning of a new view is supported by the dynamic view service. We are currently studying variations on our specifications that are more specifically tuned to systems like Isis and Ensemble. In particular, we would like to understand the power of the Isis requirement that processes that move together from one view to the next receive exactly the same messages in the first view, especially for coherent-data applications.

It would also be interesting to generalize the DVS service to dynamic *sets of primaries* rather than individual primaries, in order to allow tolerance of transient failures during a particular view.

Acknowledgments: We thank Ken Birman, who urged us to consider the interesting issues of dynamic views. We also thank Idit Keidar and Robbert van Renesse for discussions about our DVS specification and our algorithm models and proofs.

References

[1] Y. Amir, D. Dolev, P. Melliar-Smith and L. Moser, "Robust and Efficient Replication Using Group Communication" Technical Report 94-20, Department of Computer Science, Hebrew University., 1994.

[2] O. Babaoglu, R. Davoli, L. Giachini and M. Baker, "Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems", in *Proc. of Hawaii International Conference on Computer and System Science*, 1995, volume II, pp 612–621.

[3] O. Babaoglu, R. Davoli, L. Giachini and P. Sabattini, "The Inherent Cost of Strong-Partial View Synchronous Communication", in *Proc of Workshop on Distributed Algorithms*, pp 72–86, 1995.

[4] K.P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, Los Alamitos, CA, 1994.

[5] T.D. Chandra, V. Hadzilacos, S. Toueg and B. Charron-Bost, "On the Impossibility of Group Membership", in *Proc. of* 15^{th} *Annual ACM Symp. on Principles of Distributed Comp.*, pp. 322-330, 1996.

[6] F. Cristian, "Group, Majority and Strict Agreement in Timed Asynchronous Distributed Systems", in *Proc. of* 26^{th} *Conference on Fault-Tolerant Computer Systems*, 1996, pp. 178–187.

[7] D. Davcev and W. Buckhard, "Consistency and recovery control for replicated files", in *ACM Symp. on Operating Systems Principles*, n.10, pp. 87–96, 1985.

[8] D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communications", *Comm. of the ACM*, vol. 39, no. 4, pp. 64–70, 1996.

[9] D. Dolev, D. Malki and R. Strong "A framework for Partitionable Membership Service", Technical Report TR94-6, Department of Computer Science, Hebrew University.

[10] A. El Abbadi and S. Dani, "A dynamic accessibility protocol for replicated databases", *Data and knowledge engineering*, n.6, pp. 319–332, 1991.

[11] P. Ezhilchelvan, R. Macedo and S. Shrivastava "Newtop: A Fault-Tolerant Group Communication Protocol" in *Proc. of IEEE Int-l Conference on Distributed Computing Systems*, 1995, pp 296–306.

[12] A. Fekete, N. Lynch and A. Shvartsman "Specifying and using a partitionable group communication service", in *Proc. of the* 16^{th} *annual ACM Symposium on Principles of Distributed Computing*, Santa Barbara, CA, August 1997, pp. 53–62.

[13] A. Fekete, N. Lynch and A. Shvartsman "Specifying and using a partitionable group communication service", extended version of [12], available at `http://theory.lcs.mit.edu/tds`.

[14] R. Friedman and R. van Renesse, "Strong and Weak Virtual Synchrony in Horus", Technical Report TR95-1537, Department of Computer Science, Cornell University.

[15] F. Jahanian, S. Fakhouri and R. Rajkumar, "Processor Group Membership Protocols: Specification, Design and Implementation", in *Proc. of* 12^{th} *IEEE Symp. on Reliable Distrib. Systems* pp 2–11, 1993.

[16] S. Jajodia and D. Mutchler, "Dynamic voting algorithms for maintaining the consistency of a replicated database", *ACM Trans. Database Systems*, n.15(2), pp. 230–280, 1990.

[17] I. Keidar and D. Dolev, "Efficient Message Ordering in Dynamic Networks", in *Proc. of* 15^{th} *Annual ACM Symp. on Principles of Distributed Computing*, pp. 68-76, 1996.

[18] E. Lotem, I Keidar and D. Dolev, "Dynamic voting for consistent primary components", in *Proc. of the* 16^{th} *Annual ACM Symposium on Principles of Distributed Computing*, Santa Barbara, CA, August 1997, pp. 63-71.

[19] N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[20] N.A. Lynch and M.R. Tuttle, "An Introduction to Input/Output Automata", *CWI Quarterly*, vol.2, no. 3, pp. 219-246, 1989.

[21] L. Moser, Y. Amir, P. Melliar-Smith and D. Agrawal, "Extended Virtual Synchrony" in *Proc. of IEEE International Conference on Distributed Computing Systems*, 1994, pp 56–65.

[22] L.E. Moser, P.M. Melliar-Smith, D.A. Agarawal, R.K. Budhia and C.A. Lingley-Papadopolous, "Totem: A Fault-Tolerant Multicast Group Communication System", *Comm. of the ACM*, vol. 39, no. 4, pp. 54-63, 1996.

[23] J. Paris and D. Long, "Efficient dynamic voting algorithms", *Proc. of* 13^{th} *International Conference on Very Large Data Base*, pp. 268–275, 1988.

[24] R. van Renesse, K.P. Birman and S. Maffeis, "Horus: A Flexible Group Communication System", *Comm. of the ACM*, vol. 39, no. 4, pp. 76-83, 1996.

[25] A. Ricciardi, "The Group Membership Problem in Asynchronous Systems", Technical Report TR92-1313, Department of Computer Science, Cornell University.

[26] A. Ricciardi, A. Schiper and K. Birman, "Understanding Partitions and the "No Partitions" Assumption", Technical Report TR93-1355, Department of Computer Science, Cornell University.

An Adaptive Totally Ordered Multicast Protocol that Tolerates Partitions*

G. V. Chockler N. Huleihel
D. Dolev
Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel
E-mail:{grishac,nabil,dolev}@cs.huji.ac.il,
http://www.cs.huji.ac.il/{~grishac,~nabil,~dolev}

Abstract

In this paper we present a novel algorithm that implements a totally ordered multicast primitive for a *Totally Ordered Group Communication Service (TO-GCS)*. TO-GCS is a powerful infrastructure for building distributed fault-tolerant applications, such as *totally ordered broadcast*, consistent object replication, distributed shared memory, Computer Supported Cooperative Work (CSCW) applications and distributed monitoring and display applications.

Our algorithm is *adaptive*, i.e., it is able to dynamically alter the message delivery order in response to changes in the transmission rates of the participating processes. This compensates for differences among participant transmission rates and therefore minimizes fluctuations in message delivery latency. Our algorithm is thus useful for soft real-time environments where sharp fluctuations in message delivery latency are not acceptable.

Our solution provides well-defined message ordering semantics. These semantics are preserved even in the face of site and communication link failures.

1 Introduction

A group communication service with a totally ordered multicast primitive, *Totally Ordered Group Communication Service (TO-GCS)*, is a powerful infrastructure for building distributed fault-tolerant applications. Some of these are *totally ordered broadcast* [1, 8, 10, 14, 12], consistent object replication [1, 12], distributed shared memory [8], Computer Supported Cooperative Work (CSCW) applications [18] and distributed monitoring and display applications [14]. Due to its importance for distributed computing, TO-GCS has inspired a great number of research projects in universities and research institutions world-wide. Isis [3], Horus [20], Totem [2, 16], Transis [7], Amoeba [11], RMP [22], Delta-4 [17] are only some of the systems that support TO-GCS.

*This work was supported by ARPA grant number 030-7310

PODC 98 Puerto Vallarta Mexico

In this paper we present a novel total ordering algorithm for TO-GCS. Our algorithm is *adaptive*: It is able to dynamically alter the message delivery order in response to changes in the transmission rates of the participating processes. This adaptation ability compensates for differences among participant transmission rates and thus minimizes fluctuations in message delivery latency. Many soft real-time applications make certain assumptions about message delivery latency, and therefore, sharp fluctuations in message delivery latency can wreak havoc in these cases. Our algorithm is thus useful for such applications.

Another important feature of our solution is that it provides well-defined message ordering semantics. These semantics are required by existing TO-GCS based applications [1, 8, 12] and are preserved in spite of both site and communication link failures. They were first formulated within the framework of the Extended Virtual Synchrony model [15] and elaborated in [8, 12, 21]. Further discussion of our algorithm's features appears in Section 1.2.

1.1 Problem Definition

A group communication service (GCS) classically consists of two main parts: a *membership service* and a set of *multicast services*. The task of the membership service is to maintain a listing of the currently active and connected processes and to deliver this information to the application whenever the membership changes. The output of the membership service is called a *view*. The multicast services deliver messages to the current view members.

The GCS multicast service suite typically consists of a set of primitives with different ordering/reliability guarantees. The most important among these is the *totally ordered* multicast service, which guarantees to deliver messages to the current view members in a consistent order. A GCS with a totally ordered multicast primitive is called a *Totally Ordered Group Communication Service(TO-GCS)*.

In this paper we concentrate on implementing an efficient totally ordered multicast service within the group communication framework. We assume that the underlying communication layer is represented by a basic view synchronous GCS that provides membership and FIFO multicast services. The minimal requirements of the underlying GCS appear in Section 2.

The principal correctness requirements imposed by our service

are listed below. They are motivated by existing TO-GCS based applications [1, 8, 12]:

- A logical *timestamp* is attached to every message delivered by TO-GCS;
- The same timestamp is attached to a message at every process that delivers that message. This timestamp is unique system-wide and remains unique in face of network partitions;
- Every process delivers messages in the order of their timestamps;
- The timestamp order complies with the *global causal order on messages* [13], $\rightsquigarrow$, defined to be the reflexive transitive closure of the following:
  1. $m \rightsquigarrow m'$ if there exists a process p such that m was sent at p before m';
  2. $m \rightsquigarrow m'$ if there exists a process p such that m' was sent at p after m has been delivered at p.

Note that the above requirements imply that (1) any two messages are delivered in the same order at any process that delivers both of them, and (2) the message delivery order complies with the global causal order on messages.

In addition, the service implementation should satisfy the following liveness requirement:

- If a process p receives from the underlying communication layer an infinite number of messages from every operational and connected process, then p will eventually deliver every message supplied to it by the underlying communication layer or crash. (We further elaborate on TO-GCS liveness requirements in Section 3.2).

Note that the above problem is weaker in several ways than the well-known *Atomic Broadcast (AB)* problem found in the literature [10]. In particular, we do not require that each message multicast by a correct process will eventually be delivered by all correct processes; nor do we require that each message delivered by a correct process will be eventually delivered by all correct processes.

Our service is similar to the partitionable group communication service specified by the VS-machine of [8]. However, there are a few distinctions:

- Unlike [8], our service delivers application messages labeled with timestamps. The use of timestamps is motivated by the fact that TO-GCS with timestamps is useful for various TO-GCS based applications, e.g., it is utilized by the Consistent Object Replication Layer described in [12].
- The VS-machine of [8] provides safe indications, whereas TO-GCS does not. However, semantics similar to those achieved with safe indications can easily be achieved at the application level using end-to-end acknowledgments. This technique was demonstrated in [12].

1.2 Protocol Features

In this section we consider the main features of our total ordering protocol and discuss related work.

1.2.1 Dynamic Adaptation

Because a totally ordered multicast service is so useful, the efficiency of its implementation has become an important issue. A well-known technique for providing a totally ordered multicast delays delivery of a received message until the process has: (a) delivered all received messages which precede the message in question within the total order; and (b) learnt that every message that could preceded it will never arrive. This results in high latency in message delivery when not all the participant processes are uniformly active. Total ordering protocols which are based upon this technique are called *symmetric*. Another approach implemented by *sequencer* [3, 4, 11, 22] or *token* [16] based protocols uses extra messages (ordering messages or token requests) and is therefore less efficient under high loads [19].

The protocol presented in this paper is *dynamically adaptive*: Messages are assigned a wide range of priorities which are adjusted "on-the-fly" to reflect ongoing changes in process activities. Messages are then delivered in order of priority. The protocol testing results (see Section 6) show that after a short adaptation period the average message delivery latency incurred by our protocol is close to that of the underlying communication layer. Furthermore, the variance of the post-adaptation message delivery latency exhibited by our protocol is extremely low.

By contrast, under the same load patterns the latency incurred by traditional (non-adaptive) total ordering protocols is close to the transmission rate of the slowest process in the group. Moreover, these protocols exhibit sharp fluctuations in message delivery latency. This makes the message delivery latency incurred by such protocols much less predictable, causing problems for soft real-time applications. Our protocol is thus a solution for these problems.

Some systems differentiate between only two process activity levels. For example, [9] addresses the adaptivity issues by classifying group members as *active* or *passive* according to whether they have any messages to send or not; the right to multicast messages is then evenly distributed among all currently active processes. In the Hybrid protocol of [19], assignment of active or passive process status is based upon the relation between the process' transmission rate and the network delay: active processes run a symmetric protocol, while passive processes run a token-based one. Processes dynamically switch between active and passive states. The obvious limitation of the approach exemplified by these two protocols is that all the active (passive) processes are treated equally, while in practice it is rare that all of the active (passive) processes are uniformly active (passive).

In the ToTo protocol of [5] messages are delayed until messages are received from a *majority* of group members. ToTo achieves good latency only when: (a) the currently active members of the group form a majority, and (b) the processes that make up this majority broadcast their messages at approximately the same rates.

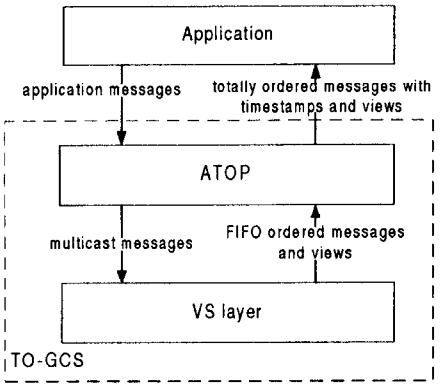


Figure 1: The TO-GCS System Components and Interfaces

1.2.2 Partionable Semantics

Additional implementation challenges are raised by the fact that the service requirements stated in Section 1.1 should be satisfied in environments where multiple concurrent network components are allowed to coexist and where processes in each component are treated as non-faulty. With many existing group communication systems [3, 5, 11, 22] the following scenario is possible: Suppose that two processes disconnect from each other, while some common non-delivered messages remain in their buffers. If these messages' order has not yet been negotiated, they may either be delivered in an inconsistent order or be discarded. Either way, the service requirements are violated.

Special care is needed to prevent such situations from occurring. Common practice [16] is to attach some ordering information to each newly multicast message. This information should be sufficient to allow each process to consistently order the message so that the need to communicate with other processes is eliminated.

Things become more complicated if the message delivery flow is allowed to be dynamically adaptive. This is because message delivery order may be altered as a result of the adaptation decision. We must therefore be careful to preserve the message delivery semantics by guaranteeing the atomicity and uniformity of the adaptation decision. This is a main challenge of incorporating adaptation into a total ordering algorithm.

The technique presented in this paper allows various external adaptation policies to be combined with the total ordering protocol. The resulting multicast service combines two valuable features: suitable performance for soft real-time applications and sound partitionable semantics.

2 The Environment Model

We assume an asynchronous distributed environment. Further, we assume that processes can fail and restart, and that the network can partition into several disjoint components which can re-merge later on. The environment is equipped with a *view-synchronous group communication layer*, called the VS (View-Synchronous) layer. This layer guarantees reliable FIFO delivery of messages that have been multicast within a group of connected and active processes. Another VS layer objective is to provide *failure detection*: Possible changes in network connectivity and in failure status of the processes are relayed via special membership change reports, called *views*. The layer is called *view synchronous* because messages are guaranteed to be delivered in the view in which they were originated. Our *Adaptive Total Ordering Protocol (ATOP)* is built on top of the VS layer. The layer structure of the Totally Ordered Group Communication Service (TO-GCS) is depicted in Figure 1.

2.1 The VS Layer Guarantees

For the rest of this paper, we denote the following: P is a totally ordered finite set of processes; M is a message alphabet; $(I, <_I, i_0)$, a totally ordered set of view identifiers with initial view identifier i_0; $views = I \times 2^P$, the set of pairs consisting of a view identifier together with a set of processes; If $v \in views$, we write $v.id$ and $v.set$ to denote the view identifier and set components of v respectively.

We define the *current view* at a process p to be as follows: if the VS layer has delivered any views at p, then the current view at p is the last such view; otherwise, it is a pair consisting the distinguished initial view identifier i_0 and the process universe P. We say that a message m is sent (delivered) in a view v at p if m is sent (delivered) at p when the current view at p is v.

The VS layer is required to satisfy the following requirements:

View Identifier Identity: Views with different process sets have different identifiers.

Initial View Identifier Uniqueness: The identifier of any view delivered by the VS layer at any process differs from the initial view identifier i_0.

Local View Identifier Monotony: Views are delivered in the view identifier order at each process .

Self Inclusion: For any view v delivered by the VS layer at a process p, $p \in v.set$.

Message Integrity: For any message m delivered at a process p in a view v, there is a preceding send event at some process q. Moreover, m is sent in v at q.

No duplication: Every message delivered by the VS layer at a process p is delivered only once at p.

Reliable FIFO Delivery: For any two messages m, m', processes p, q, and a view v: If m is sent before m' in v and q delivers m', then q delivers m before m'.

3 The TO-GCS Specification

3.1 Correctness

In addition to the message ordering properties outlined in Section 1.1, we require that TO-GCS satisfies the following:

View Properties are similar to the first four guarantees provided by the VS layer (see Section 2.1);

Basic Message Delivery Properties:

1. For every message delivered by TO-GCS to the application there is a preceding send event. Furthermore, this send event occurs in a view whose identifier is not greater than that of the view in which the message is delivered;
2. Each message is delivered in the same view at any process at which the message is delivered;
3. The sender of a message is always a member of the view in which the message is delivered.

3.2 Liveness

We require of TO-GCS to satisfy the same liveness specifications as those guaranteed by the VS layer. Since the liveness specification of the VS layer is out of the scope of this paper (the interested reader may refer to [8, 21]), we only require that for every process p, ATOP at p preserves the liveness semantics provided by the underlying VS layer. More precisely, we require the following:

1. Every application message sent through ATOP is eventually transferred to the underlying VS layer unless a crash occurs;
2. If the VS layer delivers an infinite number of messages to ATOP from every member of the current view, then ATOP will eventually deliver every message that the VS layer has passed to it, or crash;
3. If the VS layer informs ATOP about a new view v, then ATOP will eventually deliver v or crash. Moreover, ATOP at p is bound (unless it crashes) to eventually deliver every message that the VS layer has transferred to it before v.

The first two properties together ensure that if all members of the current view keep sending messages, then ATOP at a process p preserves every liveness guarantee provided by the VS layer at network stability periods. For example: if, during the network stability periods, applications at all processes in the current stable component send infinite number of messages and the VS layer guarantees to deliver all messages sent through it (as required in [8]), then TO-GCS also guarantees to deliver every application message.

Note that the requirement that every process in the current view should issue an infinite number of messages may seem unrealistic. We require it only for the sole purpose of simplifying the protocol presentation. In the actual implementation, this precondition can be enforced by ATOP itself: it can simply multicast special *dummy* messages when its application becomes "silent" (see discussion in Section 5).

The third property ensures that if the VS layer provides additional liveness guarantees at times of new view installations, then ATOP will preserve them.

4 The Adaptive Total Ordering Protocol (ATOP)

In this section we describe the Adaptive Total Ordering Protocol (ATOP) which implements TO-GCS using the VS layer. The adaptive total ordering protocol at each process consists of two modules: the module implementing an adaptive total ordering *mechanism* (ATOM) and the module implementing an adaptation *policy* (AP). Such decoupling allows various external adaptation algorithms to be easily plugged into ATOP. The service structure is depicted in Figure 2.

4.1 The AP Module

The AP module is an implementation of an external adaptation policy. It thus keeps track of messages and views delivered by the VS layer, in order to learn about the transmission rate distribution among the current view members. From time to time (depending on the adaptation policy implemented) AP at p delivers a distribution, $dist$, to the ATOM module. A *distribution* is defined to be a pair consisting of: the *distribution identifier*, $dist.id$, taken out of a totally ordered set of distribution identifiers $(D, <_D, d_0)$ with an initial distribution identifier d_0; and a vector, called the *weights vector* and denoted $dist.w$, with an entry for each $q \in P$ such that $\sum_{q \in P} dist.w[q] = 1$.

Let v be the current view at a process p. We require that the following be satisfied by every distribution $dist$ delivered by AP at p in v:

- for each $q \in v.set$, $dist.w[q] \neq 0$, and for each $q \notin v.set$, $dist.w[q] = 0$;
- Let $dist'$ be a distribution delivered by AP at q in v. If $dist'.id = dist.id$, then $dist'.w = dist.w$. This means that every distribution delivered at any process in the same view has a unique identifier;
- $dist.id \neq d_0$.

4.2 Message Ordering Using Distributions

The ATOM module controls the message ordering using two distributions: the first one, called the *sending distribution*, is used to tag each newly multicast message; and the second one, called the *ordering distribution* is used to order incoming messages. In Section 4.3 we describe in detail how these distributions are maintained.

In addition, ATOM at each process has a copy of a pre-defined pseudo-random number generator G. This generator along with the current ordering distribution's weights vector fix a deterministic sequence of process identifiers. Messages tagged with a distribution

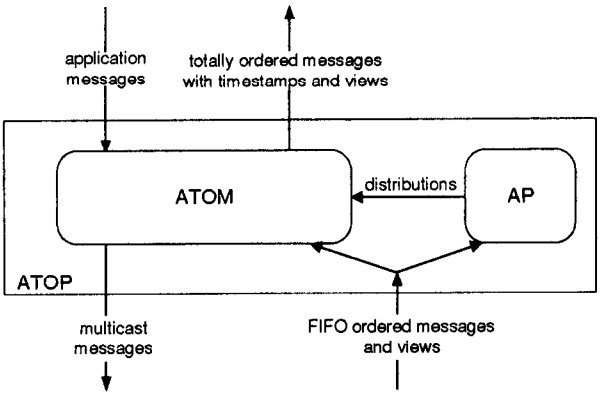


Figure 2: The ATOP Implementation

$dist$ are delivered when the current ordering distribution is equal to $dist$ and ordered by the sequence generated by G and $dist.w$.

This is illustrated by the following example: Let processes p, q and r be members of the current view. Assume that the sequence produced by G and the current ordering distribution's weight vector starts with $p, q, p, q, p, p, r, p, \ldots$. (Note that in this sequence, p apparently has a larger weight than r).

According to this sequence the first slot in the total order for the current view should be reserved for p's message. p's turn will be skipped only if the next pending p's messages is tagged with a different distribution; or if there are no more undelivered messages sent by p and there is a new view delivered by the VS layer. Likewise, the second slot in the total order should be reserved for q's message, the third slot again for p's message, and so on. Thus, for each slot, the protocol either waits for a message from the appropriate process or guarantees that no such message can be delivered in this slot, in which case it is skipped.

A detailed desctiption of the totally ordered delivery algotrithm is given in the next section.

4.3 The ATOM Module

The responsibilities of the ATOM module are as follows:

- To associate application messages with distributions;
- To guarantee that the delivery order of messages is determined by the distributions associated with them;
- To preserve the semantics provided by the underlying VS layer (see Section 2.1) in all aspects concerned with view delivery, the relative order of messages and views, and the reliable FIFO order within each view. This facilitates the achievement of the View and Basic Message Delivery properties of TO-GCS (outlined in Section 3);
- To extend the VS layer's FIFO delivery order within each view to the total delivery order in that view, so that the ordering semantics of Section 1.1 are satisfied.

ATOM associates distributions with messages by giving each newly multicast message a tag consisting of the maximal identifier among all distributions known to this instance of ATOM. The AP modules at participating processes must therefore make sure that the distribution with the maximal identifier corresponds to the most recent process transmission rate distribution. The adaptation policy described in Section 6 achieves this by allowing the AP module at only a single process (within the current view) to inject new distributions. This process is chosen deterministically from among the current view members. Other possible ways to implement the AP module are discussed in Section 7.

Within each view, message delivery order is determined by the distribution with the minimal identifier among all distributions attached to this view's yet-undelivered messages. This distribution is called the *ordering distribution*.

ATOM guarantees that the next message to be delivered is tagged with the current ordering distribution identifier. Since the AP module guarantees that every distribution has a unique identifier within each view (see Section 4.1), this means that the delivery order of each message is determined by the same distribution at any process that delivers this message.

Thus, within each view, every message is delivered in the same order at all processes that deliver this message (even at those processes that may become disconnected). Furthermore, since views are delivered in the same order at every process and each message is always delivered in the same view, ATOM guarantees the *global* total delivery order of messages.

The message timestamp assigned by ATOM is thus a triple consisting of the identifier of the view in which the message is delivered, the identifier of the current ordering distribution and the sequence number of the message within the current ordering distribution. We can easily see that the timestamp assigned in such a way satisfies the TO-GCS ordering requirements (see Section 1.1):

1. The timestamp is globally unique because: (1) the VS layer View Identifier Identity property guarantees that each view has a unique identifier, and (2) every distribution has a unique identifier within each view;
2. Each message has the same timestamp at every process that delivers this message because: (1) each such process delivers the message in the same view; (2) within each view, the message is assigned the distribution identifier when it is initially

sent; and (3) within each view, messages which are stamped with the same distribution identifier, are delivered in the same order;

3. Messages are delivered in the order of their timestamps because: (1) the VS layer Local View Identifier Monotony property ensures that views are delivered in the order of their identifiers, and (2) the identifier of the current ordering distribution increases monotonically within each view;

4. Each message m' sent by p after the delivery of another message m cannot be delivered before m at p. Therefore, when m' is delivered it is given a timestamp greater than that of m. Since ATOM preserves the VS layer's FIFO delivery order, thus the timestamp order satisfies causality.

Finally, to satisfy the TO-GCS liveness requirements, ATOM should not arbitrarily deliver any new views, nor should it arbitrarily change the current ordering distribution: instead, ATOM may deliver a new view only after it has validated that no more new messages belonging to the last view delivered to the application will ever arrive, and it may change the current ordering distribution only after it validates that no more messages tagged with the current ordering distribution will ever be received in the current view.

If this is not observed, then the following situation is liable to arise: suppose that a message m belonging to a view v were to arrive after a newer view had been delivered to the application. In this situation, correctness can only be preserved if we discard m. This, however, violates the liveness requirements. A similar situation will occur if a message tagged with some distribution identifier arrives after the current ordering distribution has been reset to a newer value.

We utilize the VS layer's Message Integrity property in order to tell that no more messages will be delivered by the VS layer in some view. This property implies that after the VS layer delivers a view v, it will not deliver any message sent in any view delivered before v in the future.

We make use of the VS layer's Message Integrity and Reliable FIFO delivery properties in order to verify that no more messages tagged with some distribution identifier d will be delivered in the current view. These properties imply that: if for each member of the current view the VS layer has delivered either (1) some message tagged with a distribution identifier greater than d or (2) a new view, then the VS layer will never deliver any message stamped with d within the current view.

A detailed description of the ATOM module algorithm is given below.

Sending Messages

The ATOM module at a process p learns about new distributions either directly from p's AP or from messages sent by ATOMs at other processes.

Let v be the current view at p. We define the *sending distribution* at p to be the distribution with the maximal identifier among distributions received at p in v, if any, otherwise it is a distribution $dist_v$, called the *default distribution for* v, such that $dist_v.id = d_0$ and $dist_v.w[q] = 1/|v|$ for each $q \in v.set$, and $dist_v.w[q] = 0$ otherwise.

The following attributes are attached by the ATOM module at a process p to each newly multicast application message:

- *sender*: the p's identifier;
- *dist_id*: the identifier of the current sending distribution at p;
- *seqno*: the sequence number of this message within the current sending distribution at p.

The first message to be tagged with a distribution's identifier will also bear the weights vector component of this distribution. (This is in addition to the above mentioned values.)

Basic Message and View Delivery

ATOM buffers messages and views delivered by the VS layer. Views are stored in the set $PendingViews$. Messages delivered by the VS layer in a view v are stored in the set $PendingMsgs[v.id]$.

Views delivered by ATOM are taken from the $PendingViews$ set. The next view chosen for delivery is the view with the minimal identifier among the views currently in the $PendingViews$ set. Delivered views are deleted from the $PendingViews$ set. Obviously, since the VS layer guarantees to deliver views in the order of their identifiers, the same is true for ATOM as well.

If v is the last view delivered by ATOM to the application, then the next message to be delivered is taken from the $PendingMsgs[v.id]$ set. ATOM suspends delivery of new views as well as of messages sent in these views until both conditions shown in Figure 3 are true.

1. $PendingViews \neq \emptyset$;
2. $PendingMsgs[v.id] = \emptyset$;

Figure 3: The Conditions for the New View Delivery

Since the VS layer guarantees that each message m is delivered in the same view at every process that delivers m, then there exists a view identifier i such that $m \in PendingMsgs[i]$ at any process that received m from the VS layer. Thus each ATOM module that delivers m to the application, delivers m in the same view.

Message Delivery within a View

Let Num be an enumeration of process identifiers in P. Let w be a weights vector as defined in Section 4.1 and G_w be a pseudo-random number generator which produces $Num(r)$ with probability $w[r]$ on each invocation. We denote a pseudo-random number obtained on the ith invocation of G_w by $G_w(i)$, $i \geq 0$.

The ATOM module at each process p maintains the following data structures:

- *ordDist* holds the current ordering distribution. It is initialized to be the default distribution for (i_0, P);

- $next[q]$ is the total number of messages sent by q which have been ordered by $ordDist$. Initially, $next[q]$ is set to be 0 for each $q \in P$.

  Clearly, the sum of $next[q]$ for each $q \in P$ holds the total number of messages that have been ordered by $ordDist$ so far. We define $totalOrdered \stackrel{\text{def}}{=} \sum_{q \in P} next[q]$.

- TS is a timestamp attached to every message delivered to the application. At any state of the protocol TS is defined to be a triple consisting of the current view identifier, $ordDist.id$ and $totalOrdered$ variables. The order on timestamps is lexicographic;

- G is an instance of a pseudo-random number generator known by all processes in P. Initially, G is set to be $G_{dist_{(i_0,P)}.w}$ and is initialized to some predefined seed agreed upon by all processes in P.

Let v be the view which was most recently delivered by ATOM to an application at a process p, if any, or (i_0, P) otherwise. We first consider how $ordDist$ is maintained. $ordDist$ is initially set to be the default distribution for (i_0, P). Whenever ATOM delivers a new view to the application $ordDist$ is set to be the default distribution for this view. Within each view $ordDist$ is reassigned a new distribution when all the conditions depicted in Figure 4 are true.

1. $PendingMsgs[v.id]$ does not contain any message tagged with $ordDist.id$;
2. There is some message $m \in PendingMsgs[v.id]$ such that $m.dist\_id > ordDist.id$;
3. For each $q \in v.set$, there is a message m sent by q such that $m.dist\_id > ordDist.id$, or $PendingViews \neq \emptyset$.

Figure 4: The Conditions for Changing the Ordering Distribution

Whenever the value of $ordDist$ changes (as a result of either a new view delivery or fulfillment of conditions in Figure 4) the following steps are performed:

1. $ordDist$ is assigned that distribution whose identifier is minimal from among the identifiers of all distributions attached to the messages currently in $PendingMsgs[v.id]$. Note that because the VS layer guarantees the reliable FIFO delivery within a view, there is always a message in $PendingMsgs[v.id]$ which contains the new distribution's weights vector;
2. $next[q]$ is set to 0 for each $q \in P$;
3. G is set to be $G_{ordDist.w}$ and is initialized to some seed agreed upon by all members of v.

The ATOM module delivers a pair (m, TS) on the next invocation of its delivery procedure iff the conditions sketched in Figure 5 are satisfied. Note that these conditions imply that (1) the next message to be delivered (from among messages currently in $PendingMsgs[v.id]$) is determined according to the weights vector of the distribution in which this message was sent; and (2) the message delivery order is consistent with the order of message sending, i.e., the message delivery order preserves FIFO.

1. $m \in PendingMsgs[v.id]$;
2. m is tagged by $ordDist.id$;
3. m is sent by a process q such that $G_{ordDist.w}(totalDelivered) = Num(q)$;
4. $m.seqno = next[q]$.

Figure 5: The Conditions for the Delivery of (m, TS)

Whenever a pair (m, TS) is delivered to the application the following steps are performed:

1. m is removed from $PendingMsgs[v.id]$;
2. $next[m.sender]$ is incremented;

If none of the conditions in Figures 3, 4 and 5 are satisfied, ATOM blocks unless the conditions in Figure 6 are true. These conditions indicate that no more new messages stamped with $ordDist.id$, which were sent by a process q such that $G_{ordDist.w}(totalDelivered) = Num(q)$ in v, will ever be received from the VS layer. We can therefore try to deliver another message in $PendingMsgs[v.id]$ labeled with $ordDist.id$ (if such a message exists).

1. $PendingMsgs[v.id]$ does not contain any message m sent by a process q such that $G_{ordDist.w}(totalDelivered) = Num(q)$;
2. There is a message $m' \in PendingMsgs[v.id]$ sent by q such that $m'.dist\_id > ordDist.id$, or $PendingViews \neq \emptyset$;
3. There is another message in $PendingMsgs[v.id]$ labeled with $ordDist.id$.

Figure 6: The Conditions for Skipping the Current Timestamp

In this case, ATOM increments $next[q]$, and thus *skips* a message that could have been sent by a process q and tagged by the current values of $ordDist.id$ and $next[q]$. This way other messages in $PendingMsgs[v.id]$ which are stamped with $ordDist.id$ and have not yet been delivered, get a chance to be delivered in one of the successive delivery attempts.

Finally, if all the conditions in Figures 3, 4, 5 and 6 are false, ATOM blocks until the VS layer delivers either a new message or a new view, which would in turn cause one of the aforementioned conditions to become true.

5 Remarks on the ATOP Performance Guarantees

There are two important issues that were intentionally left out of consideration in the ATOP protocol definition in the previous sec-

tion: They are *flow control* and *failure detection*. This simplification allowed us to better concentrate on subtleties of achieving adaptive total ordering in partitionable environments. However, this inevitably weakened our performance claims.

For example, since changing the ordering distribution requires a message tagged with a greater distribution identifier from each member of the current view (see Figure 4), a single process (or a group of processes) that has no application messages to send may substantially slow down switching to the new ordering distribution. In this case, an appropriate flow control mechanism will enforce each such "silent" process to issue a *dummy* message tagged with a new distribution identifier, thus speeding up the agreement.

Unfortunately, in distributed asynchronous systems with failures there are situations in which no flow control algorithm can help much. In particular, the performance of ATOP depends in great extent on how fast the underlying VS layer delivers messages and how fast faulty processes are removed from the view. For example, if the VS layer fails to guarantee timeliness of failure detection, the ATOP protocol may be subject for significant delays during network instability periods.

The above problems can be addressed by combining the adaptation policy, failure detection and flow control mechanisms together within the same module. For example, the failure detector can use distributions produced by the AP module to guarantee that each process either transmits messages in the rate corresponding to its weight or is taken out of the current view.

Thus, for example, an instance of the failure detector at a process p will take care of situations in which p has no application messages to send by issuing special *dummy* messages in the rate corresponding to the current p's weight. Subsequently, if an instance of the failure detector at q fails to hear messages from p in the rate which roughly corresponds to the current p's weight it will suspect p and initiate the view change. Note, that since the adaptation policy is based on application messages (and not on *dummy* ones), this mechanism would not affect the adaptation decision.

6 The ATOP Implementation and Performance Results

In order to evaluate the performance of ATOP, we implemented a simple adaptation policy. This is described in Section 6.1 below. The resulting protocol was implemented over the Causal Multicast Service (CMS) of the Transis GCS [7] which satisfies the VS layer correctness specifications presented in Section 2.

6.1 An Adaptation Policy Implementation

In the adaptation policy we implemented, only the AP module at a single process deterministically chosen among the current view members, called a *book-keeper*, has the right to inject new distributions. The book-keeper's algorithm is as follows.

Let v be the current book-keeper's view. The book-keeper maintains a *sliding window* of messages delivered by the VS layer in v. The size of the window is $N \cdot |v.set|$, where $N \in \mathcal{N}^{>0}$ is the protocol's parameter called the *window size factor*. Let $N[r]$ denote the number of messages sent by a process r from among the messages currently in the sliding window.

Let ϵ be a small positive real number. The book-keeper maintains a vector, $weights$, with an entry for every process in P such that at any instant, $weights[r] = (N[r]+\epsilon)/|v.set|(N+\epsilon)$ if $r \in v.set$, and 0 otherwise. Thus, the $weights$ vector approximates the distribution of the process transmission rates among the members of $v.set$. The parameter ϵ is needed to avoid assignment of zero weights to the $v.set$ members. Note that $\sum_{r \in v.set} weights[r] = 1$ at all times.

The $dist\_no$ variable counts distributions that have been output in the current view. Whenever a new distribution is output, $dist\_no$ is incremented and the content of the $weights$ vector is saved in another vector called $last\_weights$. If no distribution has yet been produced, $last\_weights[r] = 1/|v.set|$ for each $r \in v.set$, and 0 otherwise. Periodically, the distributions stored in $weights$ and $last\_weights$ are compared. If the difference between these distributions exceeds a predefined threshold, a distribution $dist$ such that $dist.id = dist\_no$ and $dist.w = weights$ is output.

6.2 The Testing Environment

We tested our protocol on 6 *Pentium-120* machines running the *BSD/OS* operating system and connected by a 10 MBit/second Ethernet LAN. Of these, two machines multicast at a rate of approximately 10 messages/sec and 1 machine that multicast at a rate approximately 20 messages/sec. The remaining 3 machines multicast at substantially lower rate which varied from one experiment to another. The message size was 50 bytes. During the testing period all the machines were connected and active. The observed message loss was negligible.

A potential weakness of this testing environment is that the transmission rates of participating machines was preset in advance and was static during each experiment. In the future we intend to analyze the performance of our protocol in more dynamic settings (see Section 7).

6.2.1 Performance Results Analysis

In our experiments we compared the performance of ATOP with a non-adaptive symmetric total ordering protocol, *All-Ack* [6], as well as with the Transis CMS. The Transis CMS guarantees only that the message delivery order satisfies the causal partial order on messages. Thus, in the Transis CMS, message order should not be agreed upon by all processes before delivery. Therefore, The Transis CMS (in the absence of message loss) has an average message delivery latency close to that of the underlying network, as well as a low message delivery latency variance. We thus chose the results of the Transis CMS message delivery latency analysis as references for the best achievable by any total ordering protocol.

In the first experiment series, we ran the All-Ack, ATOP and Transis CMS protocols while in each new experiment the transmission rate of each slow machine was smaller than it was in the previous experiment. We observed that (1) in each experiment the average message delivery latencies incurred by ATOP after adaptation and Transis CMS were close to one another (the average latency of ATOP was slightly greater); and (2) the latency of the All-Ack protocol steadily increased from one experiment to another.

The conclusion is that the average latency of All-Ack, unlike that of ATOP, varies according to the transmission rate of the slowest process and therefore cannot be predicted in advance.

In the second group of experiments we fixed the transmission rate of each of the slow processes to be approximately 1 message every 3 sec, and measured the variance in the message delivery latency. To do this, we compared the delivering rate of messages sent by a particular process, with the sending rate of those same messages. We observed for each of the tested protocols, that while the average delivery rate for messages sent by each process is close to the transmission rate for that process, the message delivery rate variance of All-Ack (see Figure 7(b)) is much greater than that of ATOP (after the adaptation) (see Figure 7(a)).

Figure 8(b) illustrates that in the All-Ack protocol, the message delivery blocks until a message from the slowest process arrives. Then, all pending messages are delivered at once. By contrast the post-adaptation message delivery rate of ATOP (see Figure 8(a)) is almost constant and close to the sender's transmission rate.

7 Other Adaptation Policies

The adaptation policy described in Section 6 is suited for LAN environments, where all connected processes see more or less the same picture. The same is not true for wide area networks, where the variance in the message round trip time among different processes might be significant, and different processes do not necessarily observe the same distribution for process transmission rates. Here, it is not a good idea to give the book-keeping responsibilities to a deterministically chosen process.

A better adaptation policy would instead dynamically reassign book-keeping responsibilities, while taking into account inter-process round trip delay times. The process with the minimal variance of inter-process round trip delays would obviously be the best candidate for current book-keeper.

Further challenges are presented by scenarios in which one or more participating processes may occasionally pause and then resume communication before being taken out of the current view. Clearly, such perturbing processes can easily cause the adaptation policy of Section 6 to not stabilize. An adaptation policy that would result in better performance would thus identify such perturbing processes and assign them weights which would rapidly decrease the influence of their past transmission activity (e.g., one can use the exponential backoff technique).

A completely different approach is to make the adaptation policy application dependent. For example, the application can specify possible message transmission rate distribution patterns in advance. The adaptation policy can thus recognize these patterns earlier and correspondingly change the current ordering distribution. In particular, this is useful in environments where message transmission rate distribution pattern depends on the time of day.

Acknowledgments

We are thankful to Dahlia Malkhi, Gil Neiger, Yehuda Afek and anonymous referees for their valuable comments which helped us to substantially improve the presentation quality. Idit Keidar greatly contributed to this work with her original ideas and insight. We would like to thank Tal Anker and Ohad Rodeh for suggesting interesting testing ideas. We are grateful to Aviva Dayan for proof reading our drafts and helping to improve the overall presentation quality.

References

[1] AMIR, Y. *Replication Using Group Communication Over a Partitioned Network.* PhD thesis, Institute of Computer Science, The Hebrow University of Jerusalem, Israel, 1995.

[2] AMIR, Y., MOSER, L. E., MELLIAR-SMITH, P. M., AGARWAL, D. A., AND CIARFELLA, P. The totem single-ring ordering and membership protocol. *ACM Trans. Comp. Syst. 13*, 4 (November 1995).

[3] BIRMAN, K. P., SCHIPER, A., AND STEPHENSON, P. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comp. Syst. 9*, 3 (1991), 272–314.

[4] CHANG, J. M., AND MAXEMCHUCK, N. Realiable Broadcast Protocols. *ACM Trans. Comput. Syst.* 2, 3 (August 1984), 251–273.

[5] DOLEV, D., KRAMER, S., AND MALKI, D. Early Delivery Totally Ordered Broadcast in Asynchronous Environments. In *23rd Annual International Symposium on Fault-Tolerant Computing* (June 1993), pp. 544–553.

[6] DOLEV, D., AND MALKI, D. The design of the transis system. In *Theory and Practice in Distributed Systems: International Workshop* (1995), K. P. Birman, F. Mattern, and A. Schipper, Eds., Springer, pp. 83–98. Lecture Notes in Computer Science 938.

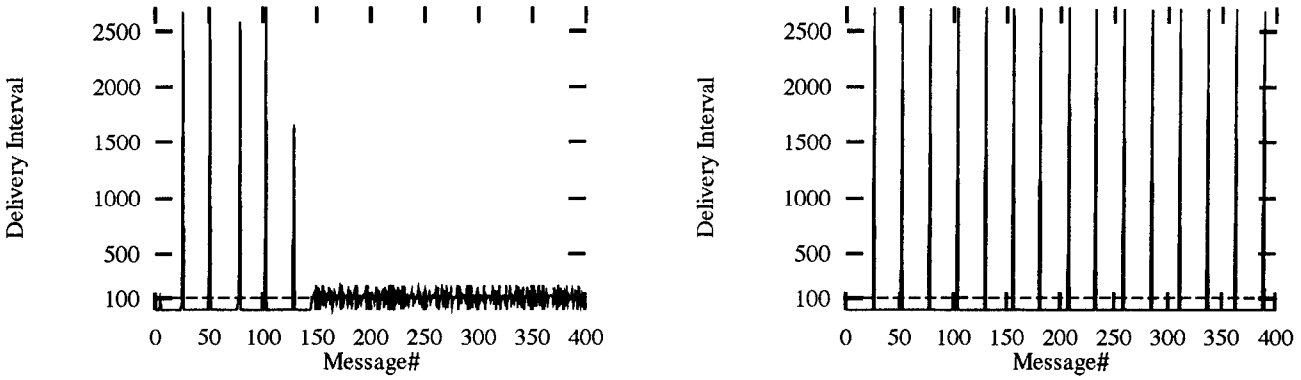
[7] DOLEV, D., AND MALKI, D. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM 39*, 4 (April 1996).

[8] FEKETE, A., LYNCH, N., AND SHVARTSMAN, A. Specifying and Using a Partionable Group Communication Service. In *16th Annual ACM Symposium on Principles of Distributed Computing* (August 1997).

[9] FRIEDMAN, R., AND VAN RENESSE, R. Packing Messages as a Tool for Boosting the Perfomance of Total Ordering Protocols. TR 95-1527, dept. of Computer Science, Cornell University, August 1995.

[10] HADZILACOS, V., AND TOUEG, S. Fault-Tolerant Broadcasts and Related Problems. In *chapter in: Distributed Systems*, S. Mullender, Ed. ACM Press, 1993.

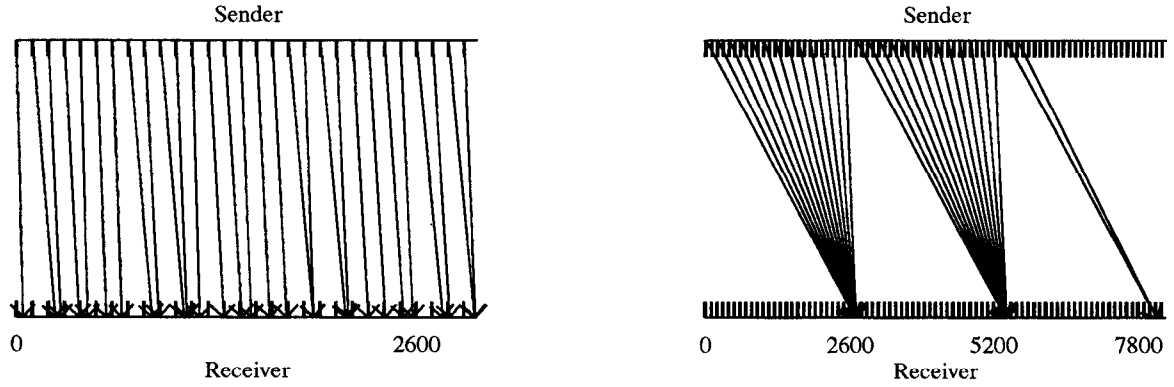
[11] KAASHOEK, M. F., AND TANENBAUM, A. S. An evaluation of the Amoeba group communication system. In *Proceedings of the 16th International Conference on Distributed Computing Systems* (May 1996), pp. 436–447.



(a) The Post-Adaptation Delivery Rate of ATOP

(b) The Delivery Rate of All-Ack

Figure 7: Delivery Rates for Messages Sent at ~10 msgs/sec



(a) ATOP After Adaptation

(b) All-Ack

Figure 8: Message Delivery Rates vs Transmission Rates for Messages Sent at ~10 messages/sec (1 $tick \sim 100\ msec$)

[12] KEIDAR, I., AND DOLEV, D. Efficient Message Ordering in Dynamic Networks. In *15th Annual ACM Symposium on Principles of Distributed Computing* (May 1996), pp. 68–77.

[13] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM 21*, 7 (July 78), 558–565.

[14] MALKI, D. *Multicast Communication for High Avalaibility.* PhD thesis, Institute of Computer Science, The Hebrow University of Jerusalem, Israel, 1994.

[15] MOSER, L. E., AMIR, Y., MELLIAR-SMITH, P. M., AND AGARWAL, D. A. Extended Virtual Synchrony. In *Intl. Conference on Distributed Computing Systems* (June 1994). Also available as technical report ECE93-22, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

[16] MOSER, L. E., MELLIAR-SMITH, P. M., AGARWAL, D. A., BUDHIA, R. K., AND LINGLEY-PAPADOPOULOS, C. A. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM 39*, 4 (April 1996).

[17] POWELL, D. *Delta-4: A Generic Architecture for Dependable Distributed Computing.* Springer-Verlag, 1991.

[18] RODDEN, T. A survey of CSCW systems. *Interacting with Computers 3*, 3 (1991), 319–353.

[19] RODRIGUES, L. E. T., FONSECA, H., AND VERISSIMO, P. Totally ordered multicast in large-scale systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems* (May 1996), pp. 503–510.

[20] VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. Horus: A Flexible Group Communication System. *Communications of the ACM 39*, 4 (April 1996).

[21] VITENBERG, R., KEIDAR, I., CHOCKLER, G. V., AND DOLEV, D. Group Communication System Specifications: A Comprehensive Study. Tech. rep., Inst. of Comp. Sci., The Hebrew University of Jerusalem, 1997. In preparation.

[22] WHETTEN, B., MONTGOMERY, T., AND KAPLAN, S. A high perfomance totally ordered multicast protocol. In *Theory and Practice in Distributed Systems: International Workshop* (1995), K. P. Birman, F. Mattern, and A. Schipper, Eds., Springer, pp. 33–57. Lecture Notes in Computer Science 938.

Supporting Quality Of Service in HTTP Servers

Raju Pandey J. Fritz Barnes Ronald Olsson
Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis, CA 95616
{pandey, barnes, olsson}@cs.ucdavis.edu

Abstract

Most implementations of HTTP servers do not distinguish among requests to different pages. This has the implication that requests for popular pages have the tendency to overwhelm the requests for other pages. In addition, HTTP servers do not allow a site to specify policies for server resource allocation. This paper presents a notion of *quality of service* that enables a site to customize how an HTTP server should respond to external requests by setting priorities among page requests and allocating server resources. It also describes a design and an implementation of a distributed HTTP server, QoS Web Server, that enforces the quality of service constraints. The performance analysis of the prototype server indicates that the server provides the desired quality of service with minimal overhead.

1 Introduction

With the advent of the WWW [13], there has been a fundamental shift in the way information is exchanged among systems connected to the Internet. Three elements [26] of the WWW make this possible: a uniform naming mechanism (URL) for identifying resources, a protocol (HTTP) [2] for transferring information, and the client-server based architecture [17]. A client such as a browser uses the URL of a resource to locate an HTTP server that provides the resource. It then requests for services associated with the resource. The HTTP server performs the requested services (such as fetching a file or executing a program) and returns the results back to the client.

The architecture of HTTP servers has been studied in great detail and different variations of HTTP servers have been proposed. Much of the work has focussed on addressing the performance limiting behaviors [22] of HTTP servers. The research has, thus, focussed on developing techniques (such as information caching [7, 20, 9, 23] and distribution, partitioning [16] of server load across clients and servers, and parallelization [15, 4, 14, 18] of HTTP servers over SMPs and workstations) for eliminating performance bottlenecks arising due to the lack of sufficient CPU, disk, and network bandwidths as well as inherent limitations in the implementation techniques of HTTP servers.

While this has led to a deeper understanding of how HTTP servers operate when there are sufficient resources for various requests, not much work has been done in cases when HTTP servers are overwhelmed by the sheer volume of requests. The behavior of HTTP servers is quite unpredictable in such cases: they either completely bog down with pending requests resulting in unacceptable response times or start to drop requests indiscriminately. In addition, requests for popular pages have the tendency to overwhelm the requests for other, possibly more important, pages. Addition of new resources (such as new machines) may not solve the problem as requests for the popular page may continue to overwhelm other requests. Further, most implementations of HTTP servers treat all requests uniformly. A site, thus, cannot assign priorities to different pages or control how its server resources should be used. For instance, a site may wish to state that a set of specific pages (such as its main page or product page) be always available irrespective of the demands for other pages or that only 20% of its resources be allocated to anonymous `ftp` requests.

One possible mechanism for ensuring that requests for a collection of pages are guaranteed some server resources is to physically separate pages from each other by hosting them on separate servers. The problem with this approach is that it is difficult to map allocation of resources to various requests statically. First, such allocation may not be precise. Second, it may lead to inefficient utilization of server resources. Third, the granularity of such partitioning can be applied only to large groups of pages.

What is needed is a notion of *quality of service* (QoS) that characterizes the behavior of an HTTP server given a set of requests. This paper presents such a notion for HTTP servers and describes a design and an implementation of an HTTP server, QoS Web Server, that enforces the proposed quality of service model. Specifically, this paper addresses the following:

- *What is an appropriate quality of service model for HTTP servers?* The quality of service model presented in this paper is aimed at enabling a site to customize how an HTTP server should respond to external requests. This includes setting priorities among page requests, allocating different kinds (absolute and relative) server resources to different requests, and specifying constraints such as "always" which indicate that a specific page (or groups of pages) should always be available.

PODC 98 Puerto Vallarta Mexico

- *How can such HTTP servers be implemented?* An implementation requires creation of a resource model for determining various resources that exist at any given moment. The paper describes an algorithm for scheduling various requests given a resource model such that the QoS constraints are satisfied.
- *What is the performance behavior of such servers?* We are interested in characterizing the execution behavior and responsiveness of HTTP servers. The results from the prototype server indicate that the implementation provides the desired quality of service with little overhead.

This paper is organized as follows: Section 2 presents a quality of service model for HTTP servers. Section 3 describes an abstract model of an HTTP server that implements this QoS model. It also includes the description of a distributed HTTP server that we have implemented. Section 4 presents an analysis of the performance behavior of the server. Section 5 contains a comparison of our work with related work. Finally, Section 6 summarizes the results and discusses future work.

2 A Quality of Service Model for HTTP Servers

The notion of quality of service has been addressed in great detail in the network and multi-media community [24]. Within a client-server framework, we can think of quality of service as a quantification of level of services that a server can guarantee its clients. Examples of typical parameters that servers have used to guarantee services are transmission delay, network transfer rate, image resolution, video frame rate, and audio or video sequence skew, among others. Clearly, the quality of service parameters depend on the kind of services that a server provides. In this section, we develop a model of quality of service for HTTP servers.

In traditional quality of service models, the emphasis has been on developing notions of service guarantees that a server can provide to its clients. For HTTP servers, we develop two views of the quality of service: client-based and server-based. In the client-based view, the HTTP server guarantees specific services to its clients. Examples of such quality of service are a server's guarantees on lower bounds on its throughput (for instance, number of bytes/second) or upper bounds on response times for specific requests. In the server-based view, the quality of service pertains to implementing a site's view of how it should provide certain services. This includes setting priorities among various requests and limits on server resource usages by various requests. We develop the QoS model by first constructing a model of client requests.

We model web pages as objects and requests to access pages as method invocations on pages. For instance, an invocation `<page>.read(`$p_1, p_2, \ldots p_n$`)` denotes a request to read `<page>`. $p_1, p_2, \ldots$ and p_n denotes parameters of the request. An HTTP server, therefore, can be thought of as a runtime system that manages executions of various method invocations. Traditional HTTP servers do not distinguish among different method invocations. Each method invocation is serviced in the order it is received (unless it is dropped due to resource contentions [6]). The QoS model here allows one to specify priority relationships among method invocations. Further, a site may specify a set of resource usage constraints for controlling the amount of server resources allocated to requests.

Note that the constraints over different requests can be classified into two types: server-centric and client-centric. Server-centric constraints depend on the attributes of servers only. Such constraints do not distinguish among different requests to the same page. Hence, priority is established among requests for different pages. Client-centric constraints depend on the attributes of clients as well. Here, requests for the same page are distinguished and may be provided different quality of service. Our focus in this paper is on server-centric constraints only.

As part of the QoS model, we have devised a notation, which we call WebQoSL. WebQoSL supports specifications of the following:

- Allocation of specific and relative amount of server resources to specific page requests
- Availability of groups of pages at all time
- Time-based and link-relation-based allocation of resources
- Scalable allocation of resources
- Specification of guarantees about byte transfer and page request rates

Below, we provide a brief overview of the notation informally. We emphasize that WebQoSL is still evolving as we are still experimenting with the notation by implementing different kinds of quality of service models.

2.1 Specification of server resources

WebQoSL allows one to model server resources explicitly:

- Percentage of server resources

  Notation: Let the term `<page>.server_resource` denote percent of total server resources associated with requests to `<page>`.

- Requests/second

  Notation: Let the term `<page>.num_requests` denote number of requests per second associated with `<page>`.

- Number of bytes/second

  Notation: Let the term `<page>.num_bytes` denote number of kilobytes of `<page>` transmitted per second.

2.2 Specification of QoS constraints

A site specifies how its server resources should be allocated by defining a number of resource constraints of the form:

```
<condition> => <QoSConstraint>
```

The constraint specifies that if `<condition>` is true, the `<QoSConstraint>` must hold. Boolean expression `<condition>` can include specific attributes (such as time, size, owner, client, time of last access and time of last modification) of pages in constraint specifications.

QoS constraints for various requests can be defined as absolute, relative, scalable and time-bound. Absolute constraint are used to specify specific resources that are allocated to various requests. Relative constraints, on the other

hand, allow one to assign various priorities among different requests. Scalable constraints allow QoS specifications to scale as new server resources (such as new machines) are added. Time-bound constraints allow one to specify constraints that have temporal characteristics (e.g., if page p is accessed at time t, page q should be available until time $t + \Delta t$.) Due to lack of space, we will only describe absolute QoS constraints here.

The absolute constraints are specified by allocating a specific amount of resources to various requests or putting a lower or upper bound on resources. For instance, the constraint

```
<page>.server_resource = r
```

specifies that <page> be allocated r units of resources. The constraint

```
<page>.server_resource < r
```

specifies that <page> be allocated at most r units of resources. The constraint

```
<page>.server_resource > r
```

specifies that <page> be allocated at least r units of resources. Another way of specifying a lower bound on resource allocations is to assert that a page should be available at all times.

```
<page>.available = always
```

The language also supports specification of allocation of default, equal and other scalable allocation of server resources.

Example 2.1. *(QoS Specification).* The following constraints

```
<www.commerce.com/free>.server_resource < 0.1
<www.commerce.com/paid/full>.server_resource > 0.5
```

are used to divide the server resources at `www.commerce.com` into two: free that can be given up to 10% of the server resources, and full that should be given at least 50% of the resources.

The next example specifies that a particular group of pages should always be available:

```
<www.commerce.com/index>.available = always
```

■

3 QoS Web Server

In this section, we describe an abstract model for the QoS Web Server. A distributed QoS Web Server is implemented in terms of a set of HTTP servers, each executing on a different host.

In figure 1, we show the architecture of a distributed QoS Web Server which is implemented in terms of five HTTP servers $(s_1, \cdots, s_5)$ executing on different hosts. Each server responds to user's requests by accessing files from either the local disk or remote disk through the network file system and transmitting them to the client. We assume that a client can send a request to any of the HTTP servers directly by using one of the routing mechanisms (such as the Domain Name Server's redirection [8], ONE-IP mechanism [10] and router-based redirection [11]).

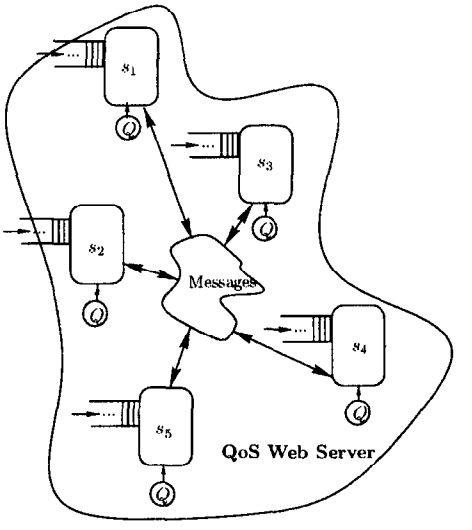


Figure 1: Architecture of a QoS Web Server

The primary goal of a QoS Web Server is to serve a file request only if servicing the requests does not violate the quality of service constraint that a site imposes. Each server, upon start, reads a file containing the quality of service specifications. It then constructs a *priority model* and a *resource requirement* model. The priority model defines a partial order among various requests to different pages and specifies the order in which requests should be handled. The resource requirements model, on the other hand, specifies the amount of resources that must be allocated to specific groups of requests. The servers then start to run and accept requests from clients.

Unlike the traditional HTTP servers where servers do not discriminate between various requests, a QoS Web Server must ensure that QoS constraints are met when requests are accepted. This is achieved by constructing a global model of resource availability and a global queue of all outstanding requests. The global resource model predicts the total amount of resources available at the hosts. The global request queue contains the pending requests. The priority model, global resource model, and global request queue are used to determine (i) the requests that will be granted service at this moment and (ii) the location of the server where a request will be executed.

We have implemented a version of a distributed HTTP server in which the global request queue and the resource model are centralized. Further, the algorithm for allocating resources is centralized as well. We describe the resource model and the HTTP server algorithm in Sections 3.1 and 3.2.

3.1 Resource model

This section briefly describes how we construct a resource model of the underlying system. The resource model specifies the capacity (in terms of bytes/second) of each HTTP server at a given moment. This provides an abstraction of resources (CPU, memory, network bandwidth) that the HTTP server can provide.

The resource model is evaluated by requiring that

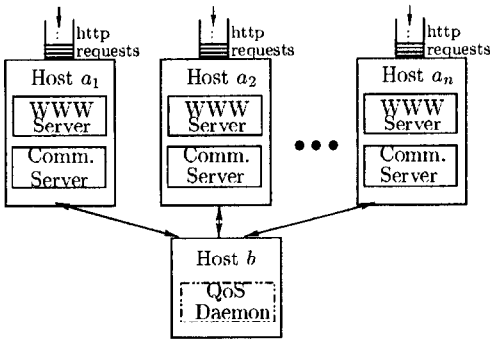


Figure 2: Architecture of the QoS Web Server

each HTTP server periodically determine the number of bytes/second it can deliver. Note that a machine's ability to serve a specific bandwidth depends on a number of factors: CPU speed, local CPU load factor, file server's capacity, file server's load factor, and local area network characteristics. In [19], an analytical model is created for evaluating the cost associated with accessing remote files, whereas in [25], the experimental technique used in the NFS benchmark (LADDIS) for evaluating the performance behavior of NSF servers is described. Both of these techniques can be extended to construct a resource model for the QoS Web Server.

Currently, we are using a simple experimental technique for constructing the resource model. We measure the length of time to send a request and use it to extrapolate the amount of bytes the HTTP servers can handle. This is performed as follows: Each HTTP server keeps a table of various local load factors and its capacity to access its local and remote files. In addition, the servers keep track of the average number of concurrent HTTP requests being served. Every time a job finishes, the table is updated and revised by calculating the transmission time. The total bandwidth is then calculated (approximately) by utilizing the average number of concurrent HTTP requests made during the interval. The average of the total bandwidth calculated by the recent jobs is then used to determine the total bandwidth for the server at a given CPU load.

3.2 An HTTP server

In this section, we describe the implementation of the QoS Web Server.

3.2.1 Architecture

We have implemented the QoS Web Server by modifying the NCSA's `httpd` web server. In figure 2, we show the architecture of the QoS Web Server. The QoS Web Server is implemented in terms of a set of components: a WWW server, a communications server and a centralized quality of service daemon (`qosd`). The WWW server is a modified version of the stand alone NCSA `httpd` WWW server [1]. It is used to handle individual HTTP requests. The modification in the NCSA server involves adding a check to ensure that a request is served only if the quality of service constraints are not violated. The modified server, therefore, sends a query to the `qosd` if the HTTP request should be handled. The `qosd` returns one of three values: handle the HTTP request, deny the HTTP request (because of QoS constraints), or redirect the HTTP request to a WWW server at a different host.

The *communication server* at a host performs two tasks: forwarding messages between the WWW Server at the host and the `qosd`, and implementing the resource model (Section 3.1). The communication server periodically transmits the WWW capacity to the `qosd` so that the `qosd` can update the global resource model. We have separated the communication server from the WWW server in order to avoid the overhead of initiating a new connection to the `qosd` every time an HTTP request is made. Also, the separation allows us to add new functionalities to the NCSA server without requiring extensive modifications in the NCSA server source code.

The *quality of service daemon* maintains global information for the distributed server and schedules HTTP requests. It maintains a quality of service model for various pages indicating priorities and resources associated with different requests, a global queue of outstanding HTTP requests, and a global resource model indicating the capacities of the WWW servers. We now describe how we use this set of information for implementing the `qosd`.

3.2.2 Implementation of the QoS daemon

The `qosd` first reads the QoS specification and constructs a QoS model. The QoS model defines categories or subsets of the document space and is used to associate an absolute or relative resource allocation with documents within the subset.

The `qosd` models each WWW server as a pipe capable of supporting a dynamic byte stream. It determines the capacity of the pipe in terms of number of bytes transmitted per second. Each WWW server periodically sends its projected capacity over the next allocation time unit to the `qosd`. Each pipe is further subdivided into smaller units, called *channels* (figure 3). A channel forms a connection between a server and a single HTTP client. It is the unit of allocation and resource control in the QoS Web Server.

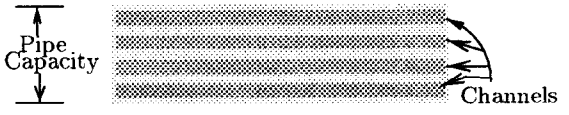


Figure 3: Pipes and channels

The size of each channel (in terms of bandwidth) is dependent on how many times we subdivide a pipe. For example, if a server indicates that it can serve 20 MB/second, the pipe size is 20 MB/second. Further, this pipe can be subdivided into 10 2 MB/second channels or 40 .5 MB/second channels. A channel with 2 Mb/second capacity is different from a channel with 0.5 MB/second capacity in that it can serve a request 4 times faster than the latter channel. The channel capacity has, thus, implications on response time. Our implementation allows a site administrator to specify the server response time for a given file of certain size[1]. The administrator can specify that a WWW page of size x

[1]The response time does not consider the latency and transmission costs across a wide area network.

should be served in time t. This can be handled by defining the channel size to be x/t.

The scheduling of HTTP requests is achieved by keeping track of two sets of requests: requests waiting to be serviced and currently being serviced. We first schedule all jobs in categories which should *always* be served. We then determine the number of remaining channels that can be allocated to requests with bounds on resource usage.

For each such category, we determine the number of channels available. We subtract from the number of available channels for this category the number of channels currently in use by requests in this category. This tells us how many channels we can allocate for new jobs in this category. We start jobs if we can start them on the server at which they arrived. After applying the algorithm, some categories may not have used all of their slots because the server at which the request arrived does not have any open channels. At this time the `qosd` redirects the request to a server with a free channel.

We assign all requests in the bounded quality of service category a lifetime. When a request surpasses a set age, QoS Web Server send a message to the HTTP client denying their HTTP request. Such a denial allows the QoS Web Server to put a limit on the implicit resources it allocates to various requests. For instance, each request occupies a space on request queue, holds a socket connection, and may even have a process assigned to it. By dropping connections, the server indicates that the request is not going to be assigned any resources in the near future as it is still trying to serve more important jobs.

4 Performance analysis

In this section, we present an evaluation of the QoS Web Server. The objectives of the evaluation are to address the following:

- How does the QoS Web Server perform for different kinds of resource constraints?
- What is the overhead of adding the notion of quality of service to HTTP servers?

4.1 Performance analysis environment

Our test environment consists of ten Sun workstations, consisting of a combination of Sparc 2, Sparc 5, Sparc 10, and Sparc 20 workstations. These workstations are connected on a local area network.

For the purpose of comparing results, we created a benchmark program based on `ptester`, a HTTP retrieval benchmark program included in the `phttpd` package [12]. The benchmark program takes as input a trace of requests and times, and uses the trace to send requests to the QoS Web Server. We generate traces that reflect specific or random mixes of various requests for different pages. All of our experiments, thus, were conducted on synthetic page requests.

The benchmark program is also responsible for calculating response times and storing the results for each request as to whether the request was accepted, was denied or failed. It allows reply of a trace of requests so that we can compare the behavior of the QoS Web Server under different configurations. The benchmark program is multi-threaded and distributed across multiple processes. This distribution is utilized in order to avoid limits due to the number of open sockets per process.

The tests were conducted on a local area network. As a result, the measurements obtained by these experiments provide a look at how to optimize the sending of pages from the Web Server's standpoint. They do not address issues related to the bandwidth of the network between the server and the clients.

4.2 Resource usage constraints

In this section, we present the set of experiments that characterize the behavior of the QoS Web Server with respect to different resource usage constraints. Specifically, our concern here is addressing the following issue: Does the QoS Web Server implement specified constraints on resource allocation to various requests? The experiments show that achieving the desired service specification depends on several facts:

- Our scheduling algorithm tries to satisfy resource constraints and, at the same time, utilize server resources effectively. Hence, if the QoS Web Server in not in contention, allocation of resources to various requests reflect the mix of the input requests. However, when the QoS Web Server is in contention, resources are allocated according to the constraints.
- In a given request mix, the QoS Web Server allocates a categories entire portion of resources only if there are enough requests in that category. For instance, the QoS Web Server can allocate 60% of its resources to requests for page A only if the requests are greater than 60% of the total QoS Web Server bandwidth.
- Channel size and request queue lifetime both affect how precisely the QoS Web Server can allocate various resources. Increasing channel size and lengthening the request queue lifetime increase accuracy but decrease response time.

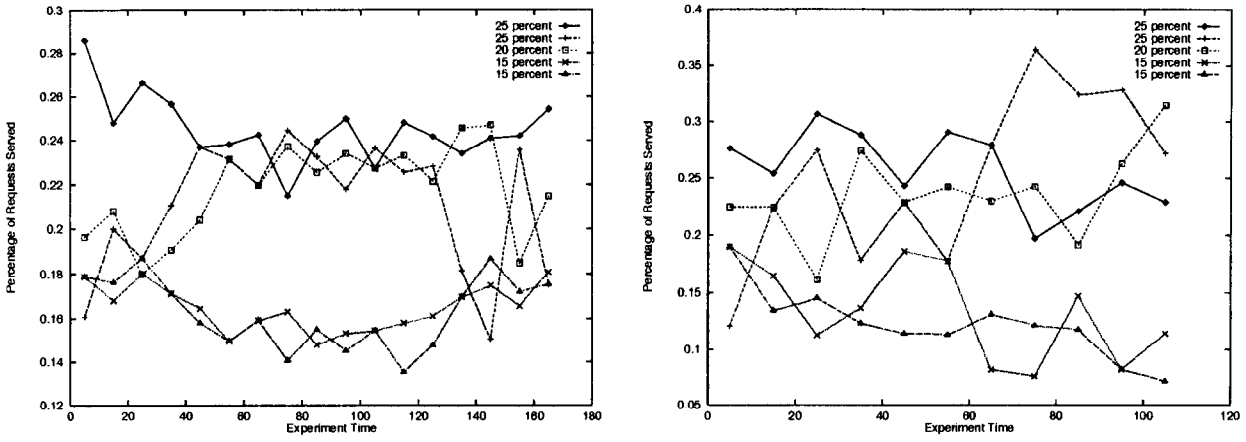
In the resource usage constraint experiments, we specify fixed percentages for jobs in a given category. We then randomly requests jobs from the different categories. Also, we utilize two to five categories of pages. We carried out the the various experiments by changing the following parameters: page size, resource usage constraints, queue lifetime and channel size.

4.2.1 Percentage requests handled

This experiment measures the number of pages served in each of the five categories over a ten second interval. We then calculate the percentage of pages served from each of the five categories.

In the first set of experiments, the benchmark program sends 18 requests per second for 16K files and 8 requests per second for 128K files. The life time for each request on the request queue was set to be 1/2 second. Figure 4(a) displays the results for files of size 16K; figure 4(b) displays the results for files of size 128K.

The graph shows the experiment time and plots the percentage of the server responses for the five different categories. The legend shows the resource constraints for various pages. As we can see, the server enforces the constraints on amount of resources that can be allocated to various pages.



(a) 16K Files

(b) 128K Files

Figure 4: Percentages of requests served for pages with different resource usage constraints

Note that there are some fluctuations in the percentage of pages served. The fluctuations arise primarily due to the randomness in the number of various category requests that arrive at the server.

4.2.2 Guaranteed service

In this experiment, we determine if the QoS Web Server can enforce resource constraints that specify that a set of pages should always be available. We request pages in five categories (A, B, C, D and E). We specify the constraint that A should always be available and that B, C, D and E receive 30%, 30%, 20% and 20% of the remaining server resources respectively.

We ran two sets of experiments: one for 16K pages and another for 128K pages. The results of the two experiments show that the QoS Web Server accepts 100% of A requests. In table 1, we show the percentages and numbers of requests accepted by the server for the two experiments.

Pages (constraints)	Experiment 1 (16K files)		Experiment 2 (128k files)	
	% served	# served	% served	# served
A (always)	100.0	704	100.0	298
B (30%)	90.0	538	59.7	172
C (30%)	84.1	530	62.2	173
D (20%)	52.6	339	42.4	123
E (20%)	54.5	354	45.5	122

Table 1: Performance behavior of server with *always* constraint

Note that the server accepts all requests for the guaranteed category. It denies about 750 requests in the 16K experiment and 500 requests in the 128K experiment for the remaining categories.

4.2.3 Different file sizes

We ran another set of experiments in order to analyze the behavior of the server when clients request files of different sizes. In this experiments, requests for files of sizes 16K, 32K, and 64K are respectively allocated 10%, 35%, and 55% of server resources.

The results of the percentages of requests handled in each of these categories are shown in figure 5(a). Instead, if we scale the results to measure the number of bytes served in each of these categories, the results appear as shown in figure 5(b).

Note that the percentage of bytes seems to match the QoS specification best. This matches our resource model that considers the resources of the server to be the bandwidth. Although, this fits better we also note that the larger file receives a disproportionate amount of the server resources. This is due to the diminishing effect of the constant overhead of making a connection to the server.

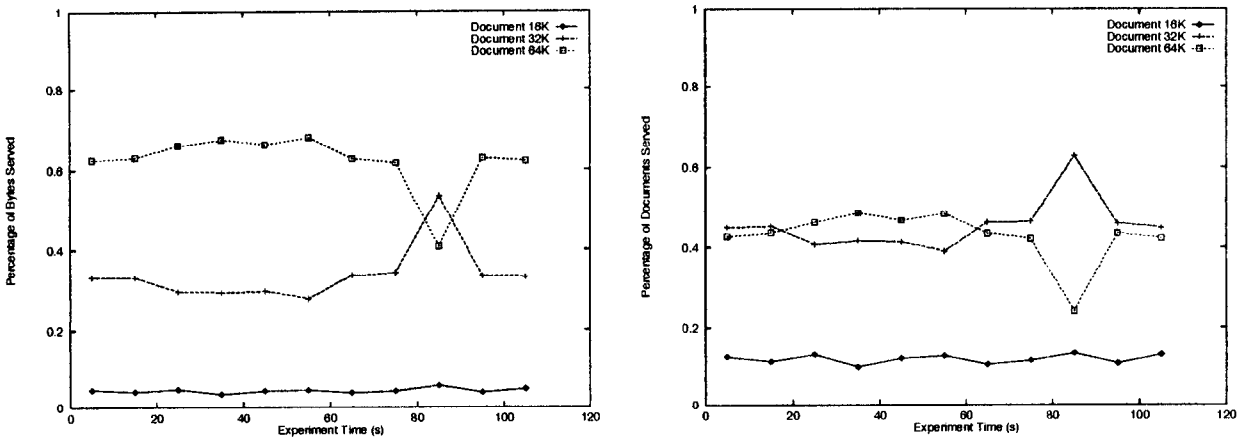
4.2.4 Flash crowds

In this experiment, we observe the behavior of the server when there is a drastic change in the number of requests for a specific page. This experiment aims to simulate the situation when there is high demand for a temporarily popular page. All file sizes are 15K and we create five categories each of which has a resource usage constraint of 20%. In this experiment, an equal number of requests arrive at the server at first. However, after 50 seconds, a large number of requests for page A arrives for the next 20 seconds. In figure 6(a), we show the request pattern for various requests.

In figure 6(b), we show the percentages of requests accepted by the server. Note that the percentages of requests served for A do not change.

4.2.5 Contention and non-contention behavior

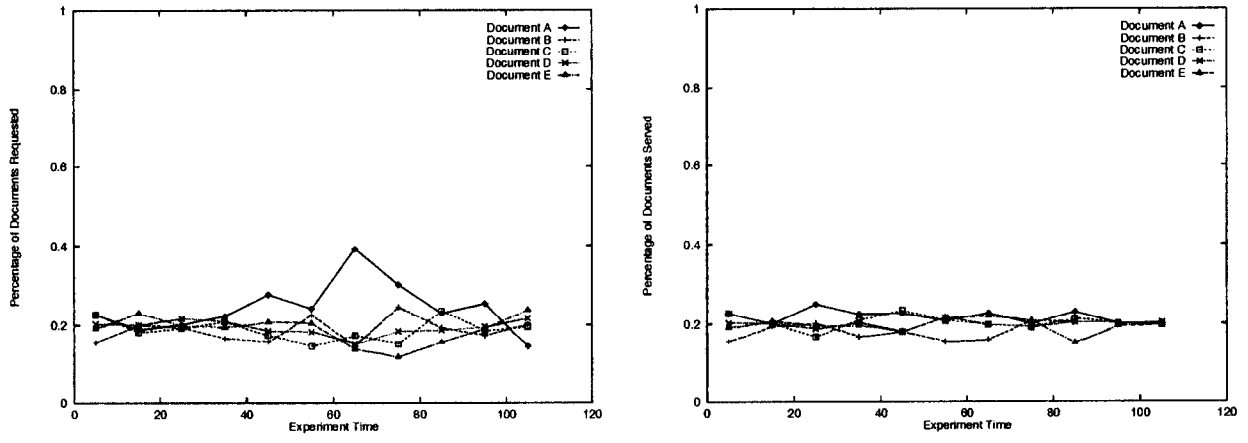
As we stated earlier, the scheduling algorithm in the QoS Web Server operates in two modes: if there is no contention, the server tries to optimally utilize resources by serving all requests. However, if there is contention, it enforces the



(a) Percentage of requests served

(b) Percentage of bytes served

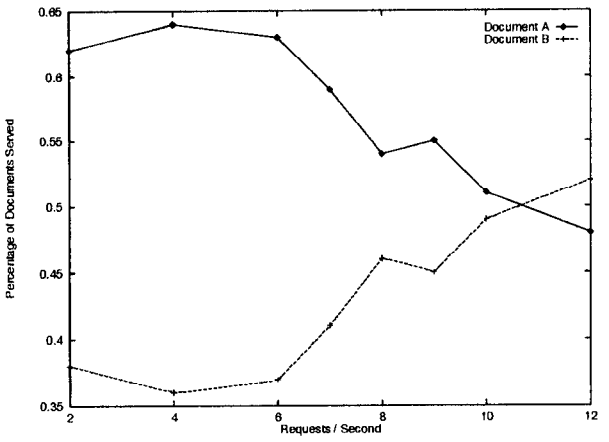
Figure 5: Behavior of server for requests of different sizes



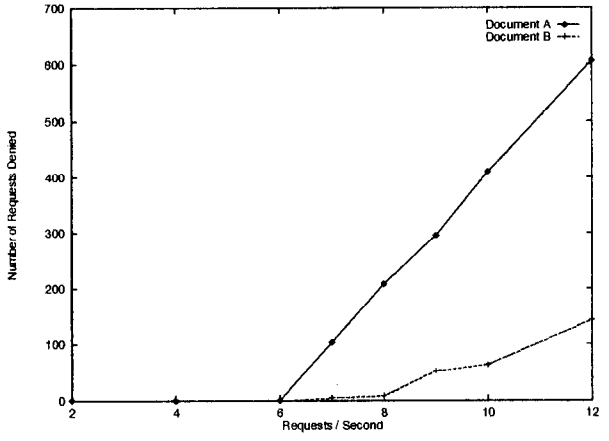
(a) Request pattern

(b) Service pattern

Figure 6: Behavior of server with flash crowds



(a) Accepted requests



(b) Denied requests

Figure 7: Behavior of server with differing number of concurrent requests

resource constraints. This set of experiments shows how the behavior of the server changes when contention arises in the server.

In this experiment, clients request two files, denoted A and B. The size of each file is 128K. The incoming requests are a mix of 65% page A and 35% page B. The QoS specification assigns equal resources to both A and B. The request lifetime for each file is 1 second. In figure 7, we show the behavior of the server.

In figure 7(a), we show the percentages of requests of A and B accepted. Note that contention begins to occur at about 8 requests/second. At about 12 requests/second, the server is in full contention. Note that as long as there is no contention, the percentages of server's acceptances of A and B match those of the requests. However, as we reach contention, the percentages of server's acceptances start to match those of the resource specifications.

In figure 7(b), we show the number of requests denied to meet the resource constraints. As long as we are not under contention, no requests are dropped. However, when in contention the server begins to deny requests in a manner that attempts to satisfy the resource constraints.

4.3 Performance comparisons

In this set of experiments, we compare the performance behavior of the QoS Web Server with respect to the NCSA HTTP server which we modified. We have compared two characteristics of the servers: throughput and average response time. In the experiments here, the tester program requests 8 files every second. The size of the files is 128K.

In figure 8, we show the throughput of the two servers. For the NCSA server, it is about 0.78 M bytes/second. The throughput for the QoS Web Server ranges from 0.42 M bytes/second to about 0.7 M bytes/second. The graph illustrates two points: First, the throughput of the QoS Web Server is only marginally less than that of the NCSA server. Hence, the overhead of adding the notion of quality of service to an HTTP server does not cause the performance of the HTTP server to degrade significantly. Second, increasing the life time of requests on the request queue increases the throughput of the QoS Web Server up to some point. When the request life time is low, QoS Web Server rejects many requests which would have been granted resources. However, by rejecting these requests, the QoS Web Server wastes all resources (such as queue space, socket overhead, process creation and deletion overhead) it devoted to the requests. However, as the requests stay on the queue longer and longer, the probability that they will be served increases more, thereby leading to better utilization of server resources.

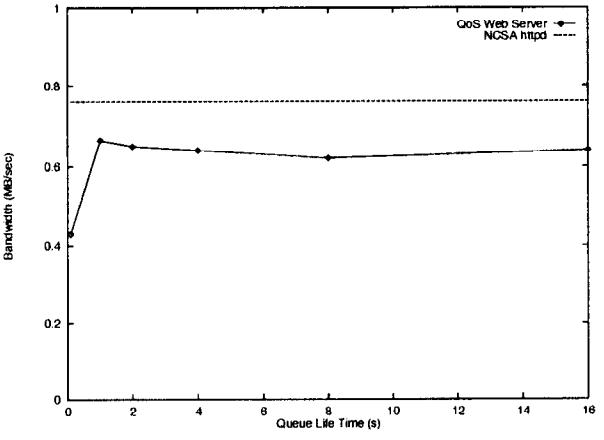


Figure 8: Comparison of throughput of NCSA and QoS Web servers

In figure 9, we show the average response times for the two servers. The lifetime for requests on the request queue is about 4 seconds. The graph highlights the fact that the average response time for the QoS Web Server remains fairly constant, whereas the response time for NCSA server is increasing. This is because the QoS Web Server drops all requests that it cannot serve after they stayed in the queue, whereas the NCSA server continues to accept requests even if it cannot handle them promptly.

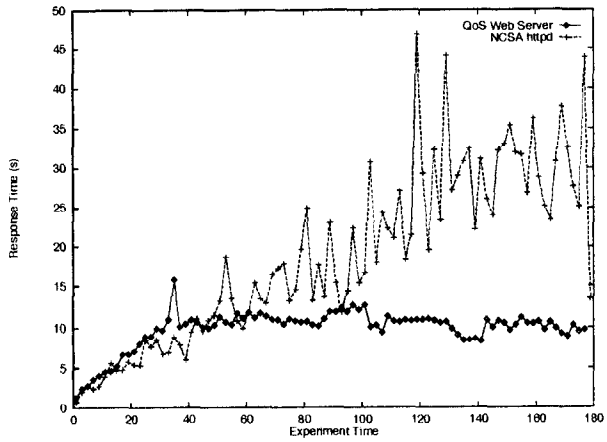


Figure 9: Comparison of average response times of NCSA and QoS Web servers

5 Related Work

There are two bodies of research with which our work overlaps: research on HTTP servers and research on quality of service in distributed systems. The focus in the first is on the design of HTTP servers, whereas the focus in the second is on developing various quality of service models and scheduling algorithms for supporting specific quality of service guarantees.

5.1 HTTP servers

As we described earlier, the primary goal of an HTTP server is to service requests for web pages. Much of the HTTP server work has focussed on developing variations of HTTP server architectures that reduce the CPU, network, and disk bottleneck. We will focus only on the distributed HTTP server work [15, 14, 4] because of the similarity in the issues addressed by these approaches and our approach. The focus in the distributed server research has been on using the resources of distributed hosts to increase the throughput of HTTP servers. Most of the research here has been aimed at addressing the notion of load balancing and scalability: given a request, how should the server schedule this request so that resources on the distributed hosts are optimally utilized. Our work, on the other hand, addresses additional issues in the design of HTTP servers:

- Should the server accept a request?
- If so, how much resources should be allocated to the request?

There has been some work that looks at the notion of quality of service for HTTP servers. [3] proposes a notion of quality of service by associating priorities with requests from different sites. The HTTP server schedules requests according to priorities, thereby ensuring that preferred sites (with higher priority) are allocated resources before other sites. Our work differs in many ways: first, the focus in [3] is on proposing techniques for structuring single host HTTP servers in order to improve the response times of high priority requests. Our work primarily involves distributed HTTP servers. Second, our notion of quality of service is more general in that we not only allow a site to specify priorities but also allow it to specify resource usage constraints on a group of requests. In [5], a notion of quality of service is proposed with respect to the content. However, there is not support for any notion of quality of service with respect to resource usage, throughput or response time.

5.2 Quality of service in Distributed Systems

The notion of quality of service [21] has been studied in great detail within the context of networking [21] and multimedia [24]. The focus of work here has been on developing varying level of services (including low-level notions such as number of bytes/second to high-level notions such as jitter-free play of images etc.) and on developing algorithms for scheduling CPU, memory and networking resources such that the quality of service guarantees are met. In [27] mechanisms for specifying service guarantees with method invocations of CORBA objects is presented. Our work is similar to these works in that we also associate quality of service with resources in order to schedule resources. However, our work differs from them in the nature of resources (web pages), in terms of constraints on usage of resource and how they should be scheduled.

6 Summary

We have presented the design and implementation of a distributed HTTP server that implements a quality of service model. In this model, a site can determine how requests for various pages should be served. This includes setting priorities among the requests as well as associating constraints on resource usages. Resource usage constraints provide a useful tool for providing services on the WWW. They support the ability to guarantee documents and set desired performance characteristics by denying requests rather than serving all requests at the same time.

We have also analyzed the performance characteristics of the QoS Web Server. The analysis shows that the server enforces user specifiable constraints on resource usages. Further, the performance behavior of the server is comparable to that of the standard NCSA HTTP server.

Our future work involves formalizing WebQoSL, refining the resource model, and implementing a distributed version of the `qosd`.

References

[1] NCSA Web Server Source. `ftp://ftp.ncsa.uiuc.edu/ Web/httpd/Unix/ncsa_httpd/httpd_1.5.2a/httpd_1.5.2a-export_source.tar.Z`.

[2] Hypertext Transfer Protocol (HTTP): A protocol for networked information. `http://www.w3.org/hypertext/WWW/Protocols`, 1995.

[3] Almeida, J., Dabu, M., Manikutty, A., and Cao, P. Providing Differentiated Levels of Service in Web Content Hosting. Tech. rep., University of Wisconsin-Madison, 1998.

[4] Andersen, D., Yang, T., Egecioglu, O., Ibarra, O., and Smith, T. SWEB: Towards a Scalable World Wide Web Server on Multicomputers. In *Proceedings of*

the Tenth IEEE International Symposium on Parallel Processing (1996), IEEE Computer Society, pp. 139–148.

[5] BANATRE, M., ISSAMY, V., LELEU, F., AND CHARPIOT, B. Providing Quality of Service over the Web: A Newspaper-based Approach. In *Proceedings of the Sixth International World Wide Web Conference* (1997).

[6] BANGA, G., AND DRUSCHEL, P. Measuring the Capacity of a Web Server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems,Monterey, California, USA* (December 1997).

[7] BESTAVROS, A. WWW Traffic Reduction and Load Balancing Through Server-Based Caching. *IEEE Concurrency* (1997), 56–67.

[8] BRISCO, T. DNS Support for Load Balancing. *Network Working Group, RFC 1794* `http://andrew2.andrew.cmu.edu/rc/rfc1794.html`.

[9] CAUGHEY, S., INGHAM, D. B., AND LITTLE, M. C. Flexible Open Caching for the Web. In *Proceedings of the Sixth International World Wide Web Conference* (1997).

[10] DAMANI, O. P., CHUNG, P. E., HUANG, Y., KINTALA, C., AND WANG, Y. ONE-IP: Techniques for Hosting a Service on a Cluster of Machines. In *Proceedings of the Sixth International World Wide Web Conference* (1997).

[11] DIAS, D., KISH, W., MUKHERJEE, R., AND TEWARI, R. A Scalable and Highly Available Server. In *COMPCON* (1996), pp. 85–92.

[12] ERIKSSON, P. The PHTTPD World Wide Web Server. `http://www.signum.se/phttpd`.

[13] ET AL., T. B.-L. The World-Wide Web. *CACM 37*, 8 (August 1994), 76–82.

[14] GARLAND, M., GRASSIA, S., MONROE, R., AND PURI, S. Implementing Distributed Server Groups for the World Wide Web. Tech. Rep. CMU-CS-95-114, Carnegie Mellon University, 1995.

[15] KATZ, E., BUTLER, M., AND MCGRATH, R. A Scalable HTTP Server: The NCSA Prototype. In *Proceedings of the First International Conference on the World-Wide Web* (May 1994).

[16] KONG, K., AND GHOSAL, D. Pseudo-serving: A User-Responsible Paradigm for Internet Access. In *Proceedings of the Sixth International World Wide Web Conference* (1997).

[17] KWAN, T. T., MCGRATH, R. E., AND REED, D. A. NCSA's World Wide Web Server: Design and Performance. *IEEE Computer* (November 1995), 68–74.

[18] LIU, Y., DANTZIG, P., WU, C. E., CHALLENGER, J., AND NI, L. M. A Distributed Web Server and Its Performance Analysis on Multiple Platforms. In *Proceedings of the The Sixteenth International Conference on Distributed Computing Systems* (1996), pp. 665–672.

[19] MELAMED, A. S. Performance Analysis of Unix-based Network File Systems. *IEEE Micro* (February 1987), 25–38.

[20] NABESHIMA, M. The Japan Cache Project: An Experiment on Domain Cache. In *Proceedings of the Sixth International World Wide Web Conference* (1997).

[21] NAHRESTEDT, K., AND SMITH, J. M. The QoS Broker. *IEEE MultiMedia Magazine* (Spring 1995), 53–67.

[22] PREFECT, F., DOAN, L., GOLD, S., WICKI, T., AND WILCKE, W. Performance Limiting Factors in HTTP (Web) Server Operations. In *COMPCON* (1996), IEEE, pp. 267–272.

[23] SCHEUERMANN, P., SHIM, J., AND VINGRALEK, R. A Case for Delay-Conscious Caching of Web Documents. In *Proceedings of the Sixth International World Wide Web Conference* (1997).

[24] VOGEL, A., KERHERVÉ, B., VON BOCHMANN, G., AND GECSEI, J. Distributed Multimedia and QoS: A Survey. *IEEE MultiMedia 2*, 2 (1995), 10–19.

[25] WITTLE, M., AND KEITH, B. E. LADDIS: The Next Generation of NFS File Server Benchmarking. In *Proceedings of the USENIX Summer 1993 Technical Conference* (June 1993), Usenix.

[26] YEAGER, N. J., AND MCGRATH, R. *Web Server Technology: The Advanced Guide for World Wide Web Information.* Morgan Kaufmann, 1996.

[27] ZINKY, J. A., BAKKEN, D. E., AND SCHANTZ, R. D. Architectural support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems 3*, 1 (1997), 1–20.