Group Communication as an Infrastructure for Distributed System Management *

Y. Amir

Department of Computer Science
The Johns Hopkins University
Baltimore MD 21218
and the NASA Center of Excellence
in Space Data and Information Sciences
yairamir@cs.jhu.edu

D. Breitgand, G. V. Chockler, D. Dolev

Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904 Israel
{davb, grishac, dolev}@cs.huji.ac.il

Abstract

In the past, system management tools for computer systems were oriented towards managing a single computer with, possibly, many users. When the networked system concept became widespread, centralized solutions such as the Network Information Service (NIS[1]) were developed to help the system manager to control a network of workstations. Today, when many sites contain hundreds of workstations, these solutions may no longer be adequate.

This paper proposes the usage of techniques, developed for group communication and database replication, for distributed cluster management. We show how group communication can be exploited to provide three classes of frequently needed operations: simultaneous execution of the same operation in a group of workstations; software installation in multiple workstations; and consistent network table management (improving the consistency of NIS)

1 Introduction

The rapid growth of distributed environments is motivated by a number of advantages provided by a distributed architecture over a centralized one. Among them are the better cost-performance ratio, potential for higher availability, provision for geographical spread of organizations, and, from the user point of view, more autonomy for users.

Unfortunately, distributed environments are harder to manage. Often they require management data to be scattered and duplicated in several sites. When system size grows, controlling the management data and keeping it consistent becomes a complex and tedious task.

This paper shows how group communication mechanisms can help in building efficient solutions for distributed system management. Our proposed architecture exploits a group communication service to minimize communication costs and to help preserve complete and consistent operation, despite of potential network partitions and site crashes. Although we focus on the Unix environment, a de-facto standard for distributed environments, the same mechanisms are applicable for other settings as well.

We address problems in three areas of distributed system management:

- *Table Management:* Examples for table management include the management of user accounts, maintenance of a unified file system and various network tables. We will show how new group communication and replication techniques can render table management services efficient, symmetric, consistent and highly available, while preserving the existing interface to these services.
- *Software Installation and Version Control:* Presently, software installation is mostly done manually by the system manager on a per-machine basis. We will show how group communication techniques can exploit the multicast and broadcast capabilities of local area networks to speed up the installation process and to minimize latency and network load during the installation process.
- *Simultaneous Execution:* It is sometimes necessary to invoke the same management command on several machines. For example, if an electricity shutdown time is expected, it might be advisable to shut down the whole system. Consequently, it is beneficial to have a method to invoke the shutdown command from one control station. We will show how such a centralized management tool can be easily constructed within our architecture.

The rest of this paper is organized as follows: the next section briefly describes existing work in the area. Section 3 describes Transis, our group communication toolkit. Section 4 presents the proposed architecture

*This work was supported by United States - Israel Binational Science Foundation, grant number 92-00189.

[1]NIS was formerly called *Yellow Pages*, but later renamed to avoid confusion with other trademarks.

for distributed system management. Sections 5-7 focus on simultaneous execution, software installation, and table management respectively, and Section 8 offers concluding remarks.

2 Related Work

In this section we briefly survey some existing solutions for distributed system management. In particular, we consider Network Information Service (NIS) as a configuration management framework and Distributed SMIT as a tool for concurrent execution of system administration tasks in a heterogeneous environment. In addition, Tivoli Management Environment (TME) is discussed as an example of a leading integrated solution for distributed system management.

Network Information Service.

The Network Information Service (NIS) [10] is supplied as a part of the operating system by all major UNIX vendors. In NIS, a collection of network tables (maps) constituting a configuration database, can optionally be replicated among a group of *servers*. Updates to the configuration database are always made at the distinguished server, termed *master*, and later may be propagated to the other servers, called *slaves* (if such exist). In this architecture, the master centralizes the configuration management, and the slaves are for higher availability and better performance.

While this architecture has proved to be successful, current implementations of NIS lack built-in facilities for guaranteeing consistency of replicas in the presence of network partitions and server crashes.

In particular, propagation of updates is not done automatically on a natural, per-update basis, but is relayed over the system administrator to be performed periodically. This may lead to undesirable temporal inconsistencies even when the system is stable. After each update, the corresponding table is completely rebuilt, and the whole table is sent over the network. Since TCP/IP is used, the number of slaves which can be reasonably employed is limited. Slaves are not allowed to propagate updates to other slaves. Thus, the system cannot reach a consistent state if the network partitions and the master is not present in a partition. If the master crashes, NIS cannot continue to operate without a complete reconfiguration.

In Section 7 we show how NIS implementation can be substantially improved while preserving all of its appealing features.

Distributed SMIT

Distributed SMIT[5] (DSMIT) presents an integrated tool for heterogeneous system management. DSMIT consists of *clients* which emit management commands to servers in a unified platform-independent syntactic form. The servers translate the commands into a platform-specific form and perform them in parallel.

DSMIT utilizes a reliability layer built on top of UDP [2]. This layer makes extensive use of *end-to-end* acknowledgments in order to cope with omission faults not handled by UDP. To monitor the status of each participating server, DSMIT clients retain the last acknowledged transmission. In contrast to this, our solution monitors the status of the whole group of management servers. The need to care about each particular target arises only upon a membership change, reported by the group communication layer.

The primary reason for DSMIT not to use group communication toolkits (such as Transis) was that efficient group communication solutions were restricted to LANs at the time of DSMIT's development. Recent work ([1, 3, 8]) demonstrates that the group communication paradigm can be effectively extended to WAN environment. As group communication technology over WAN matures, the reliability layer implemented in DSMIT will become less-effective.

Sections 5, 6 show how a group communication transport layer can be utilized for building effective and reliable solutions for problems DSMIT was designed to tackle.

Tivoli Management Environment

The Tivoli Management Environment [12] (TME) is probably the most comprehensive integrated solution for distributed system management existing today. We focus on two components of TME: Tivoli/Admin which deals with system configuration management, and Tivoli/Courier which addresses software distribution.

Both Tivoli/Admin and Tivoli/Courier use a communication toolkit, named MDist (multiplexed distribution) [11]. MDist is designed to distribute a large amount of data to a predefined set of target machines using point-to-point communication. These targets can be either end-receivers or *repeaters*, which can in turn become distributors. All participants are organized into a tree which is constructed in order to speed-up data distribution and to improve scalability.

Tivoli/Admin allows replication of certain configuration data in order to increase availability and performance. Consistency among the different copies is maintained using the two phase commit protocol [6]. Two phase commit performs end-to-end acknowledgment between all of the replicas for each update. Therefore, it is resource consuming and achieves limited performance which deteriorates linearly as the number of replicas increases.

While the concepts developed by Tivoli certainly constitute an integrated and comprehensive distributed system management solution, its infrastructure can be improved. Thanks to the open design of TME, our solutions can be integrated into it and coexist with other approaches.

3 The Transis System

Transis [3] is a group communication sub-system developed at the Hebrew University of Jerusalem. Transis supports the *process group* paradigm in which

[2] TCP could not be used as a transport layer because it limits the number of simultaneously opened connections and, hence, limits system scalability.

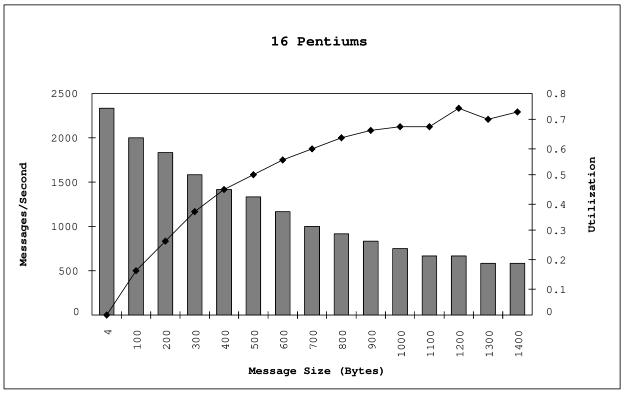

Figure 1: Throughput as a function of message size.

a process can join groups and multicast messages to groups. Using Transis, messages can be addressed to the entire process group by specifying the group name (a character string selected by the user). The group membership changes when a new process joins or leaves the group, a processor containing processes belonging to the group crashes, or a network partition or re-merge occurs. Processes belonging to the group receive configuration change notification when such an event occurs.

Transis incorporates sophisticated algorithms for membership and reliable ordered delivery of messages that tolerate message omission, processor crashes and recoveries, and network partitions and remerges. High performance is achieved by utilizing non-reliable broadcast or multicast where possible (such as on local area networks). Transis performance can be seen in Figure 1.

Transis application programming interface (API) provides a connection oriented service that, in principle, extends a point-to-point service such as TCP/IP to a reliable multicast service. The API contains entries that allow a process to *connect* to Transis, to *join* and *leave* process groups, to *multicast* messages to process groups, to *receive* messages and to *disconnect*.

Transis is implemented as a daemon. The Transis daemon handles the physical multicast communication. It keeps track of the processes residing in its computer which participate in group communication, and also keeps track of the computer's membership (i.e. connectivity). The benefit of this structure are significant. The main advantages in our context are:

- Message ordering and reliability are maintained at the level of the daemons and not on a per group basis. Therefore, the number of groups in the system has almost no influence on system performance.
- Flow control is maintained at the level of the daemons rather than at the level of the individual process group. This leads to better overall performance.
- Implementing an open group semantics is easy (i.e. processes that are not members of a group can multicast to that group).

A process is linked with a library that connects it to the Transis daemon. When the process connects to Transis, an inter-process communication handle (similar to a socket handle) is created. A process can maintain multiple connections to Transis. A process may voluntarily join specific process groups on a specific connection. A message which is received can be a regular message sent by a process, or a membership notification created by Transis regarding a membership change of one of the groups to which this process belongs. Transis service semantics is described in [2, 9].

Transis is operational for almost three years now. It is used by students of the Distributed Systems course at The Hebrew University and by the members of The High Availability Lab. Several projects were implemented on top of Transis. Among them were highly available mail system, two types of replication servers and several graphical demonstration programs.

Ongoing work on the Transis project focuses, among other things, on security and authentication of users which is important for useful distributed system management tools.

4 The Architecture

The architecture is composed of two layers as depicted in Figure 2. The bottom layer is Transis, our group communication toolkit, which provides reliable multicast and membership services. The top layer is composed of a management server and a monitor. Although we use Transis as our group communication layers, other existing toolkits such as Totem [4], Horus [13] or Newtop [7] could have been used.

The management server provides two classes of services: long-term services and short-term ones. Long-term services provide consistent semantics across partitions and over time. They are used for replication of network tables (maps) such as the password database, which are maintained on a secondary storage. These services implement an efficient replica control protocol that applies changes on a per-update basis.

Short-term services are reliable as long as the network is not partitioned and the management server

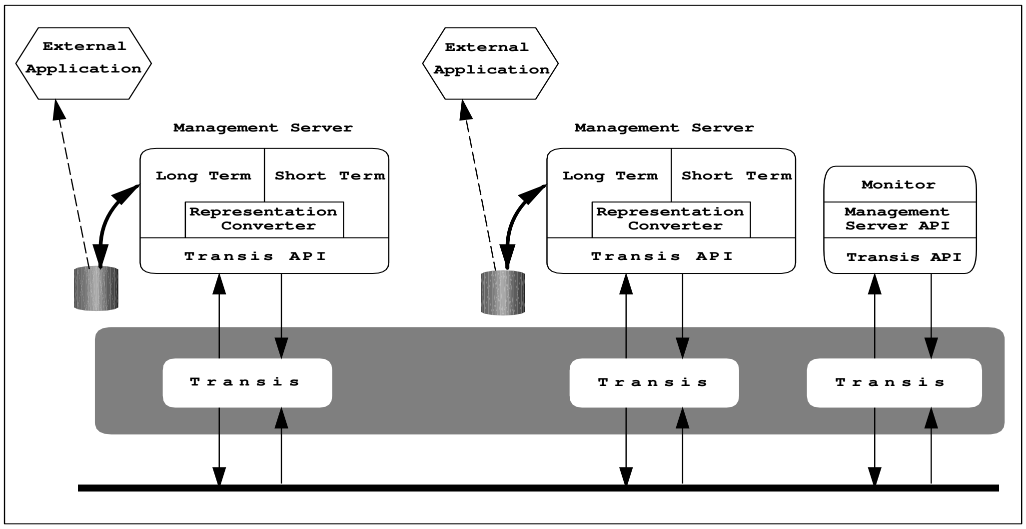

Figure 2: Two levels architecture.

does not crash. In case of a network partition or a server crash, the monitor and the management servers receive notification from Transis. The application may be informed and may take whatever steps necessary. Short-term services include simultaneous task execution and software installation.

The monitor provides a user interface to the services of the management server. The monitor is a process which may run on any of the nodes that run Transis. Several monitors may run simultaneously in the network.

The management server runs as a daemon on each of the participating nodes. It is an event driven program. Events can be generated by the monitor, another server or Transis.

Each server maintains a vector of *contexts*, with one entry for each monitor it is currently interacting with. Each entry contains (among other things) the current working directory of the server as set by the corresponding monitor.

The long-term services are a non-intervening extension of the current standard Unix NIS. Since the hosts NIS map repositories retain their original format, applications (*e.g. gethostbyname*) that use RPC to retrieve information from them are not changed, The service quality is improved because the replication scheme implemented by the management server guarantees consistency and is much more efficient compared to the ad-hoc solution provided by NIS.

The management server API contains the following entries:

- Status. Return the status of the server and its host machine.
- Chdir. Change the server's working directory which corresponds to the requesting monitor.
- Simex. Execute a command simultaneously (more or less) on a number of specified hosts. The command is executed by each of the relevant servers relatively to the working directory that corresponds to the initiating monitor.
- Siminst. Install a software package on a number of specified hosts. The installation is performed relatively to the working directory that corresponds to the initiating monitor.
- Update-map. Update map while preserving consistency between replicas.
- Query-map. Retrieve information from a map.
- Exit. Terminate the management server process.

In practice, sites may be heterogeneous both in terms of software (e.g. operating system) and hardware. We make use of a generic platform-independent representation for management commands and for the reports of their execution. This representation is the only format used for communication between the protocol entities. The Representation Converter (see Figure 2) is responsible for converting this generic representation into a platform-specific form. This architecture enables the support of new platforms with a relative ease.

A prototype of the presented architecture was implemented on top of Transis and was tested in a cluster of Unix workstations. The code, developed in the C programming language, spans approximately 6500 lines. The table management protocol (the more sophisticated part) constitutes about half of the code.

5 Simultaneous Execution

The system manager may frequently need to invoke an identical management command on several machines. Potentially, the machines may be of different types. The activation of a particular daemon or script on several machines, or the shutdown operation

(a) The Monitor	(b) The Management Server

```
Initially:
  CONNECT to Transis;
  JOIN private group;
  JOIN group Cluster;
while(true) {
  m = RECEIVE();
  switch(m.type)
    case command from a user:
      NR = 𝓜;
      MULTICAST(command, Cluster);
      while(NR ≠ ∅)
        m = RECEIVE();
        switch(m.type)
          case view change message:
            NR = NR \ (𝓜 \ m.𝓜);
            𝓜 = m.𝓜;
          case result of execution from server:
            NR = NR \ server;
      return the result;
    case view change message:
      𝓜 = m.𝓜;
}
________
command can be one of the following:
  Chdir, Status, Simex or Exit
```

(a) The Monitor

```
Initially:
  CONNECT to Transis;
  JOIN private group;
  JOIN group Cluster;
while(true) {
  m = RECEIVE();
  switch(m.type)
    case Chdir(dir) from the monitor M:
      contexts[M].working_dir = dir;
      send ACK to M's private group;
    case Status from the monitor M:
      get status of my machine;
      convert status to a system-independent form;
      send status to M's private group;
    case Simex(command) from the monitor M:
      convert command to a system-specific form;
      chdir(contexts[M].working_dir);
      result = execute(command);
      convert result to a system-independent form;
      send result to M's private group;
    case Exit:
      terminate my process;
}
```

(b) The Management Server

Figure 3: The Simultaneous Execution Protocol

of several machines are good examples. Another example is the simultaneous monitoring of CPU load, memory usage and other relevant system parameters on all or part of the machines in a cluster.

Figure 3(a) and Figure 3(b) present the pseudocode of the relevant parts of the management server and the monitor respectively. The management server maintains two sets: $\mathcal{M}$ and NR. $\mathcal{M}$ is the most recent membership of the group **Cluster** as reported by Transis. NR is the set of the currently connected management servers which have not yet report the outcome of a command execution to the monitor.

It is easy to see how other tasks are integrated with the simultaneous execution task to form our proposed architecture.

6 Software Installation

Software installation and version upgrade constitute one of the most time-consuming system management tasks. In large heterogeneous sites which comprise tens or even hundreds of machines, there are often subgroups of computers with identical (or similar) architecture running copies of the same application software and operating system. Presently, it is a common practice to perform installation or upgrade by repeating the same procedure at all locations in the subgroup separately. Installation or upgrade procedures include the transfer of the packages, the execution of installation utilities and the update of relevant configuration files. Traditionally, all the above mentioned operations are performed using the TCP/IP protocol. This approach is wasteful in terms of both bandwidth and time, due to the point-to-point nature of TCP/IP. In addition, repeating the same procedure many times is prone to human errors resulting in inconsistent installations.

In contrast, we use Transis to disseminate the relevant files to the members of the subgroup efficiently. We use the technique presented in Section 5 to execute the installation commands simultaneously at all the involved locations. Each command is submitted only once, reducing the possibility of human errors. Using the process group paradigm, the system administrator can dynamically organize hosts with the same installation requirements into a *single* multicast group.

Our installation protocol proceeds in the following steps. First, the monitor multicasts a message advertising the installation of a package P, the set R_p of its installation requirements (*e.g.* disk space, available memory, operating system version etc.), the installation multicast group G_p and the target list T_p. Upon reception of this message, the management server joins G_p, if the system which it controls conforms to R_p and belongs to T_p. When all the management servers from T_p have either joined G_p or reported that they will not participate in the installation, the monitor begins multicasting the files comprising the package P to the group G. Finally, the status of the installation at every management server is reported to the monitor. The Transis membership service helps detecting hosts which may not have completed the installation due to a network partition or host crash.

The same protocol may later be repeated for a more restricted multicast group $G' \subset G$. The monitor questions the members of G' about the missing files prior to the redistribution, and only the needed files are multicast to G' in order to save bandwidth and time.

7 Table Management

This section presents the protocol for efficient and consistent management of the replicated network tables, each of which represents a *service*. Servers which share replicas of the same table form the same *service group (SG)*. A service group consists of an administratively assigned *primary* server and a number of *secondary* ones. For the sake of simplicity we will consider a single SG in the following discussion.

The primary server enforces a single total order on all the update messages inside the SG. This is achieved by forwarding each new request for update from a client to the SG's primary. The primary creates an update message from the request, assigns it a unique sequence number, and multicasts this update message to the SG. Each secondary server applies the update messages to the SG's table in the order consistent with the primary's one. This guarantees that all the servers in the same network component remain in a consistent state. If the network partitions, at most one component (the one that includes the primary) can perform new updates. Therefore, conflicting updates are never possible.

When a membership change (network partition or merge, or server crash) is reported by the group communication layer, the connected servers exchange information and converge to the most updated consistent state known to any of them. Note that this happens even if the primary is not a member of the current membership. The information exchange is done in two stages. In the first stage, the servers exchange state messages containing a vector, representing their knowledge about the last update known to each server. In the second stage, the most updated server multicasts updates that are missed by any member of the currently connected group. [3]

Each server logs all the update messages from the primary on a non-volatile storage. This log is used for restoring of a server's state when a server recovers from a crash. A server discards an update from the log when it learns that all the other servers have applied this update to their table (and hence, no server will need to recover that update in the future).

Data Structures

Each management server $S \in SG$ maintains the following data structures:

- *my_id*: a unique identifier of S.
- *p_id*: the identifier of SG's primary server.
- $\mathcal{MQ}$: a list of the updates received by S. $\mathcal{MQ}$ is retained on a non-volatile storage.
- *Vec*: a vector of sequence numbers containing one entry for each of the SG's members. If $Vec[i] = n$ then S knows that server i has all the updates up to n. Initially, all Vec's entries are 0. *Vec* is retained on a non-volatile storage.
- *SGT*: the Transis group name of SG.
- *Memb*: the current membership of *SGT* as reported by Transis. This is a structure which contains a unique identifier of the membership (*memb_id*) and a set of currently connected servers (*set*).
- *ARU*: [4] a sequence number such that S knows that all the updates with sequence numbers no greater than *ARU* were received and applied to the table by all the members of *SGT*. Note that $ARU = min_{1 \leq i \leq |Vec|}(Vec[i])$.
- *min_sn, max_sn*: the minimal and maximal sequence numbers of update messages that need to be retransmitted upon a membership change.
- *Memb_counter*: variable that counts the *State* messages during the information exchange upon a membership change.

Message Types

- *Req*: a new request to perform an update to the table. This request is sent by a client to one of the servers. The update operation is stored in the *action* field of this message.
- *Upd*: an update message multicast by SG's primary to *SGT*. This message carries a unique sequence number in the *sn* field in addition to the fields of a *Req* message.
- *M*: a membership change notification delivered by Transis. This message contains the same two fields as the *Memb* structure.
- *State*: a state message which carries the *Vec* and the identifier of the sender. This message is stamped with the membership identifier of the membership it was sent in.
- *StateP*: similar to the state message which is used for garbage collection when the membership contains all the members of *SG*.
- *Qry*: a query message from a client.

In addition, a *type* field is included with each message.

The Pseudo Code

The following subsections present the pseudo-code of the table management protocol.

[3] If the primary server is present in a component, it will be the one performing the retransmission. Otherwise, one of the most updated secondary servers is deterministically chosen.

[4] *ARU* stands for "all-received-up-to"

Request from a client

The server which receives an update request from a client, forwards it to the primary server. The primary server creates an update message from this request, applies it to the SG's table and multicasts it to the group. Procedure HANDLE-REQUEST details these steps.

```
HANDLE-REQUEST(m)
{
    if (my_id == p_id) then
        Vec[my_id] = Vec[my_id] + 1;
        m.sn = Vec[my_id];
        m.type = Upd;
        append m to MQ;
        sync MQ and Vec to disk;
        apply m.action to SG's table;
        MULTICAST(m, SGT);
    else if (p_id ∈ Memb) then
        SEND(m, p_id);
}
```

Update from a server

A secondary server which receives an update message in the correct order, applies the update to the table and changes its data structures accordingly. Procedure HANDLE-UPDATE details these steps.

```
HANDLE-UPDATE(m)
{
    if (my_id ≠ p_id and
      m.sn == Vec[my_id] + 1) then
        Vec[my_id] = m.sn;
        append m to MQ;
        sync MQ and Vec to disk;
        apply m.action to SG's table;
    else
        discard m;
}
```

Membership change notification from Transis

Upon a membership change, the connected servers exchange information in order to converge to the most updated state. Procedure HANDLE-MEMBERSHIP prepares the data structures for this recovery process and multicasts a *State* message. Note that the *State* message contains Vec, representing the local knowledge regarding other servers' states.

```
HANDLE-MEMBERSHIP(m)
{
    Memb.set = m.set;
    Memb.memb_id = m.memb_id;
    min_sn = max_sn = Vec[my_id];
    Memb_counter = | Memb |;
    create a State message m';
    MULTICAST(m', SGT);
}
```

State message from a server

When a valid *State* message is received, the server updates its knowledge regarding other servers' knowledge. After all the *State*s messages have been received, the needed update messages are retransmitted by the most updated server. If the primary server is a member of the current membership, it is selected as the most updated server, otherwise the most updated secondary server with the smallest identifier is selected using the procedure MOST-UPDATED-SERVER.

Procedure HANDLE-STATE details these steps.

```
HANDLE-STATE(m)
{
  if (m.memb_id ≠ Memb.memb_id) then
     return;
  Vec = max(Vec, m.Vec);
  if (m.Vec[m.sender] < min_sn) then
     min_sn = m.Vec[m.sender];
  if (m.Vec[m.sender] > max_sn) then
     max_sn = m.Vec[m.sn];
  Memb_counter = Memb_counter − 1;
  if (Memb_counter == 0) then
     if ( MOST-UPDATED-SERVER() ) then
      for each m' ∈ MQ s.t. m'.sn > min_sn do
          MULTICAST(m', SGT);
}
```

The MOST-UPDATED-SERVER procedure presented below returns *true* if the invoking server is the most-updated-server with the minimal identifier, and *false* otherwise.

```
boolean MOST-UPDATED-SERVER()
{
    for each i ∈ Memb.set and i < my_id do
        if (Vec[i] == max_sn) then
            return false;
    if (Vec[my_id] == max_sn) then
        return true;
}
```

Garbage collection

In order to discard updates which are no longer needed, procedure COLLECT-GARBAGE is called upon the reception of either a *State* message, or a *StateP* message. The *StateP* message is sent periodically if the membership contains all the members of the SG. The reason for having the *StateP* message, is to avoid maintaining large amounts of updates that are no longer needed because each member of the SG has already applied them.

```
COLLECT-GARBAGE(m)
{
    Vec = max(Vec, m.Vec);
    new_ARU = min_{1≤i≤|Vec|}(Vec[i]);
    if (new_ARU > ARU) then
        for each m' ∈ MQ s.t. m' ≤ new_ARU do
            remove m' from MQ;
        ARU = new_ARU;
```

sync $\mathcal{MQ}$ and Vec to disk;
}

Events handling

The following is the main loop of the table management part of the management server.

```
Initially:
  CONNECT to Transis;
  JOIN group SGT;
  initialize all the Vec entries to 0;
  bring in MQ and Vec (if present) from disk;
  ARU = min_{1<=i<=|Vec|}(Vec[i]);
while(true) {
  m = RECEIVE();
  switch(m.type)
     case Req:
        HANDLE-REQUEST(m);
     case Upd:
        HANDLE-UPDATE(m);
     case Qry:
        retrieve an answer from the local table;
        send the answer to the client;
     case M:
        HANDLE-MEMBERSHIP(m);
     case State:
        HANDLE-STATE(m);
        COLLECT-GARBAGE(m);
     case StateP:
        COLLECT-GARBAGE(m);
}
```

8 Conclusion

We have presented an architecture that utilizes group communication to provide efficient and reliable distributed system management. The common management tasks of simultaneous execution, software installation and table management were addressed. The resulting services are convenient to use, consistent in presence of failures, and complementary to the existing standard mechanisms.

References

[1] Y. Amir, 1995. The Spread toolkit, Private Communication.

[2] Y. Amir. *Replication Using Group Communication Over a Partiotioned Network.* PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, 1995.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992. The full version of this paper is available as TR CS91-13, Dept. of Comp. Sci., the Hebrew University of Jerusalem.

[4] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. 13(4), November 1995.

[5] N. Amit, D. Ginat, S. Kipnis, and J. Mihaeli. Distributed SMIT: System management tool for large Unix environments. Research report, IBM Israel Science and Technology, 1995. In preparation.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, chapter 7. Addison Wesley, 1987.

[7] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995.

[8] N. Huleihel. Efficient ordering of messages in wide area networks. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, 1996.

[9] L. E. Moser, Y. Amir, P. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, June 1994.

[10] H. Stern. *Managing NFS and NIS*, chapter 2, 3, 4. O'Reilly & Associates Inc, first edition, June 1991.

[11] Tivoli Systems Inc. *Multiplexed Distribution (MDist)*, November 1994. Available via anonymous ftp from ftp.tivoli.com /pub/info.

[12] Tivoli Systems Inc. *TME 2.0: Technology Concepts and Facilities*, 1994. Technology white paper discussing Tivoli 2.0 components and capabilities. Available via anonymous ftp from ftp.tivoli.com /pub/info.

[13] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in Horus. In *Proceedings of the ACM symposium on Principles of Distributed Computing*, August 1995.