

Highly Available Monitoring System Architecture (HAMSA)

A thesis is submitted in fulfillment of
the requirements for the degree of
Master of Science

by
Gleb Shaviner

supervised by
Prof. Danny Dolev

Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel.

December, 2003

Acknowledgements

I am deeply grateful to my advisor, Prof. Danny Dolev, for his support, guidance and endless patience, which I appreciate very much.

I would also like to thank David Breitgand who introduced me into the theme, for his help, time and advice of great importance throughout all stages of this work.

Abstract

Monitoring of the environment is an essential functionality needed in any network management system. In order to achieve flexibility and scalability, network monitoring facilities are being decentralized. One of the more popular approaches to decentralizing is the usage of a multi-tier hierarchical structure, in which monitoring functionality is being distributed among the cooperating nodes in the hierarchy. However, the advantages of the distributed hierarchical solutions come at the cost of the increasingly complex *meta-management*. Meta-management refers to managing the management tools themselves. In other words, the effort required for maintaining a large-scale dependable distributed monitoring service may offset the advantages gained from its decentralization.

In this thesis, we study one of the more important aspects of meta-management of the distributed hierarchical monitoring services, the *high availability*. We propose a novel *Highly Available Monitoring Services Architecture (HAMSA)* that improves availability and dependability of the monitoring applications. HAMSA provides for their guaranteed behavior in an asynchronous network, being subject to the *general omission* failure model. In this model, messages may be delayed, lost, reordered, and duplicated by the network, hosts may crash and recover asynchronously, and transient network partitions (i.e., the independent isles of connectivity) may be formed due to the network errors.

HAMSA is a general-purpose *middleware* that simplifies deployment of the dependable monitoring applications in a multi-tiered setting. To achieve high availability with the *strong consistency semantics* for the *state-full* monitoring applications, HAMSA implements a novel primary/backup replication protocol that is especially well-suited for monitoring. HAMSA uses Group Communication Service internally, which is transparent to the clients and the monitored resources. The scalability of HAMSA is achieved through keeping the replication groups relatively small, and restricting the group communication service usage to the dedicated servers that have sufficient resources for utilizing it. HAMSA is a *complimentary* proposal, and does not require universal acceptance to be deployed.

The contributions of this work are as follows. We present HAMSA, and study the main algorithms required to implement it. We describe the implementation details of the fully functional prototype of HAMSA, and evaluate its performance theoretically and through an actual experimentation. We provide examples of important network management applications, in which high availability is important, and demonstrate that communication and processing overhead introduced by HAMSA is relatively low.

Contents

1	Introduction and Motivation.....	8
1.1	Traditional approaches in Network Management.....	8
1.1.1	Two-tier vs. Multi-tier architecture.....	8
1.1.2	Mobile Agents and Management by Delegation	10
1.1.3	Application Servers.....	11
1.2	HAMSA scope.....	11
2	Model	13
2.1	(k, Δ) -bofo semantics.....	13
2.2	Problematic scenarios	14
2.3	Problem statement.....	16
3	Architecture.....	18
3.1	Highly Available Mid-Level Managers (HA-MLMs)	19
3.2	HAMSA-compatible Components.....	20
3.3	HAMSA Messaging Service.....	22
3.4	Group Communication Service (GCS).....	23
4	HAMSA algorithms	25
4.1	Algorithms description.....	26
4.1.1	HAMSA states	26
4.1.2	HA-MLM membership change handling.....	27
4.1.3	External event handling	28
4.1.4	State Exchange.....	28
4.1.5	Garbage Collection	31
4.1.6	Load-balancing	31
4.2	Primary/backup protocol discussion.....	31
5	Implementation Highlights.....	33
5.1	Overview.....	33
5.2	HAMSA mid-tier framework.....	33
5.2.1	MLM.....	33
5.2.2	HAMSA component	35
5.2.3	HAMSA stateful object and state	36
5.2.4	HA-MLM.....	36
5.2.5	HAMSA plug-ins.....	40
5.3	HAMSA administration tool.....	40
5.4	Implementation structure	43
5.4.1	<i>admin</i> package	43
5.4.2	<i>component</i> package.....	43
5.4.3	<i>core</i> package	44
5.4.4	<i>exceptions</i> package.....	44
5.4.5	<i>group</i> package.....	44
5.4.6	<i>hamlm</i> package.....	44
5.4.7	<i>logger</i> package.....	45
5.4.8	<i>messaging</i> package.....	45
5.4.9	<i>plugins</i> package.....	46

5.4.10	<i>state</i> package	46
5.4.11	<i>transis</i> package	46
6	HAMSA component applications	47
6.1	Post-mortem failure analysis	47
6.2	Event-driven reactive monitoring	49
6.3	Usage-based IP billing	50
7	Performance evaluation	51
7.1	Trade-off analysis	51
7.2	Experiments	53
7.2.1	Test-bed setup	53
7.2.2	Messaging throughput	53
7.2.3	MLM recovery overhead	54
7.2.4	Discussion	57
8	Related work	59
9	Future Work	61
9.1	Advanced Load Balancing and QoS within HA-MLM	61
9.2	Automatic HA-MLM Construction	61
10	Conclusion	62
	Bibliography	63
	Appendix A: HAMSA Installation Guide	65
	Appendix B: HAMSA Administration Guide	67
	MLM server	67
	Administration tool	68
	Notes	68

List of Figures

Figure 1: Centristic approach.....	9
Figure 2: Distributed approach	10
Figure 3 HAMSAs Architecture.....	18
Figure 4: HA-MLMs Hierarchy.....	20
Figure 5: HA-MLM Structure.....	21
Figure 6: Membership change handling	27
Figure 7: External events handling	28
Figure 8: HAMSAs State Exchange protocol's state machine.....	29
Figure 9: HAMSAs state exchange protocol.....	30
Figure 10: HA-MLM internal structure	37
Figure 11: HAMSAs Messaging Service	39
Figure 12: HAMSAs administration tool	41
Figure 13: All MLMs are put into a single HA-MLM.....	47
Figure 14: Pair-wise Organization	48
Figure 15: Usage-based IP billing application.....	50
Figure 16: HAMSAs and monitoring communication cost as a function of the LANs number	51
Figure 17: HAMSAs communication cost per state change as a function of the required system error failure probability	52
Figure 18: Request-response roundtrip time as a function of state replication frequency.....	54
Figure 19: a) MLM recovery time; b) number of messages as a function of HA-MLM membership size.....	56
Figure 20: a) MLM recovery time; b) number of messages as a function of the hosted components number	56
Figure 21: a) MLM recovery time b) number of messages as a function of state size.....	56

List of Tables

Table 1: HA-MLM state	26
Table 2: HAMSA component state.....	27

1 Introduction and Motivation

As the networked systems rapidly grow in size, the management techniques of a traditional centralized network become insufficient. Distribution of management applications is required. The distribution, however, results in management tools being themselves very complex distributed systems prone to various failures that are not simple to handle. Numerous important network management applications such as usage-based accounting, trend analysis, performance management, fault management, and others, perform application-specific network monitoring tasks as part of their activities.

The primary target of HAMSA is to diminish the down time of the critical monitoring services by masking various network and host failures disrupting their normal operation. Writing a highly available service is difficult. Therefore, our monitoring middleware takes care of generic problems of distributed computing in a failure-prone environment, and provides a guaranteed behavior of the monitoring applications requiring a minimal effort from their developers.

In this work we describe the main building blocks of this architecture, and demonstrate its power for efficient and reliable monitoring by describing and analyzing the performance of monitoring applications implemented using HAMSA. The thesis is organized as follows: In Section 1 general background and motivation are provided. We present the HAMSA's model and problem scope in Section 2, an architecture overview in Section 3, and an essence of HAMSA algorithms in Section 4. Section 5 contains the HAMSA prototype implementation details. Performance evaluation highlights are covered in Section 7, and, finally, the discussion on the related work is given in Section 8.

1.1 Traditional approaches in Network Management

1.1.1 Two-tier vs. Multi-tier architecture

Most of the network management solutions prevalent today operate according to the rigid client/server architecture. In this architecture, a "thick" client (*manager*) communicates with the per-device "thin" servers (*agents*) via some common protocol [11] to retrieve management information and to control managed entities. All data processing and decision making are taking place at the manager's workstation. Target device agents are usually quite primitive and function only as an access to the devices' local management information. Therefore, the management process is, in fact, centralized at the manager's workstation.

The traditional two-tier approach to network management is being rapidly abandoned. This is motivated by the severe scalability and availability limitations of this approach, which is essentially centralized [11]. The fast progress that has been made in mobile code and distributed middleware technologies [8] makes more flexible architectures, such as the popular multi-tier one, both technically feasible and attractive also for the management applications [2] and [3].

As shown in Figure 1 among the more prominent problems with this approach are the following:

- Since the management agents have limited functionality and capabilities, they only instrument the access to the management data, while all the computations should be performed by the manager. Thus, large volumes of data should be transferred over the network, and the traffic overhead can be high.
- As shown in [10] reactive (*i.e.*, event-driven) monitoring schemes are far more efficient in terms of communication than polling-based ones. However, in standard management frameworks, such as SNMP, configuring application-specific threshold-driven traps is a non-trivial and not always a feasible task.
- Manager station concentrates all the management data aggregation and processing, and therefore, becomes a bottleneck as the size of the managed network increases.
- Manager station is a single point of failure, which damages general availability of the monitoring services. Although for some types of management data disconnected type of monitoring operation can be achieved [11], the disconnected monitoring operation is not available in the general case. This type of operation, however, is essential for scaling monitoring services, reducing communication overhead, and increasing survivability of management services as explained below.
- The sometimes unavoidable network distance between the management station and the network elements makes it very hard to control the elements, due to the inherent instability imposed by long control loops.

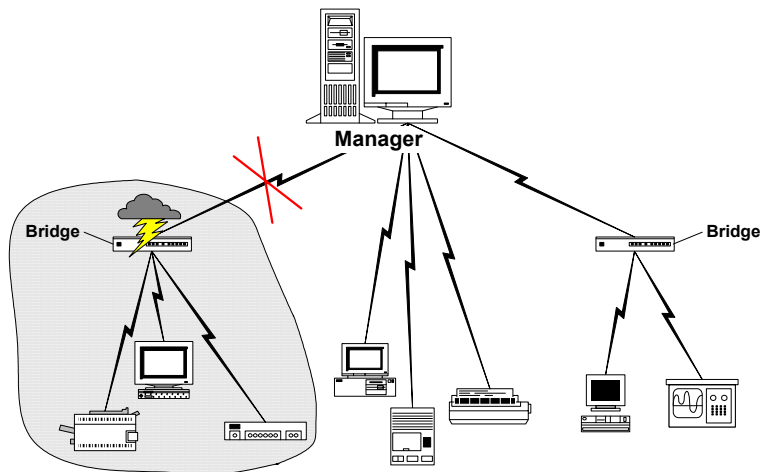


Figure 1: Centristic approach

Because of these problems, alternative approaches to monitoring architectures, such as the more flexible multi-tier one, were pursued: [15], [16], [18], [19], [20]. The more important among them, as well as their relationship to our proposal are discussed in Section 8.

A typical multi-tier monitoring application is described in Figure 15. The target agents constitute the lowest tier and serve as the source of the management data. The monitoring manager applications are dynamically dispatched at the middle tier(s), and therefore sometimes are termed *mid-level managers*. They monitor the target agents (and, possibly, communicate with other monitoring components) accumulating and pre-processing the

information collected from them. The end-consumers of this information, the management front-ends, constitute the uppermost tier.

The flexible multi-tier organization of distributed applications offers considerable benefits. Note that in the multi-tier architecture, the components residing in a middle tier can partially or fully implement some of the processing functionality that was previously residing exclusively on the manager side. Thus, using this architecture reduces the traffic overhead, shortens the control loops, and extends the management functionality. In particular, the middle-tier components can implement efficient application-specific event-driven monitoring schemes.

Survivability and availability of the network monitoring services are also improved. The mid-tier components can operate autonomously of the first-tier managers (see Figure 15). In the new scheme, the overall availability of a management application is increased since different mid-level components of it may be executed at different network locations, and a failure of one of them does not imply the immediate unavailability of the whole service. When certain parts of the monitored network become unavailable, *e.g.*, due to a network split, the mid-tier monitoring components can continue monitoring in their respective partitions, and later merge the results. This is impossible in the centralized two-tier architecture.

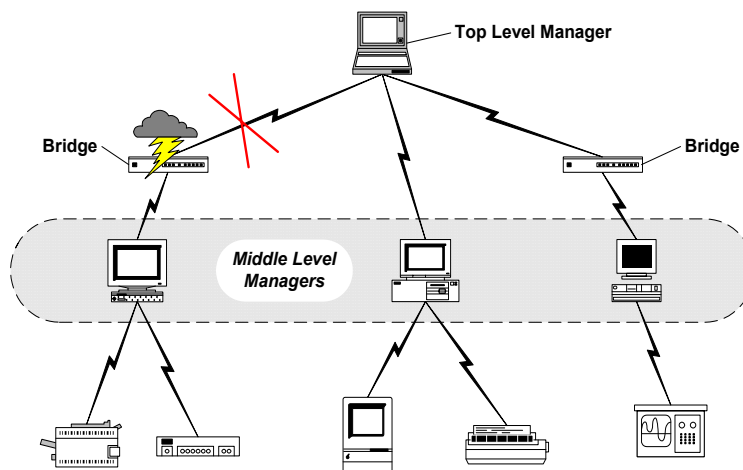


Figure 2: Distributed approach

On the down side, the multi-tier client/server applications are much more difficult to control. Providing high availability of the mid-tier components in spite of host crashes, network splits and merges is especially challenging. For example, since the dependencies usually exist among various management applications, as well as among components of the same application, even a single failure of a critical component may bring the whole management application to a halt, or render it inaccurate.

1.1.2 Mobile Agents and Management by Delegation

Further decentralization of network management can be achieved through mobile agents' approach, possibly combined with the management by delegation.

In fact, even general purpose distributed mobile agents frameworks require introducing sophisticated mechanisms for efficient management, as well as for high availability and reliability. Most of the existing frameworks, such as [13], refer to the agents' reliability by providing simple mechanisms for state backup/recovery that utilize locally available non-volatile storage, *e.g.*, a file system.

With regard to the network management, reference [2] proposes a distributed management framework based on the mobile agents' paradigm. References [3] and [4] propose a combination of mobile agents and management by delegation approaches. Reference [5] proposes distributed management with mobile agents, while retaining the standard SNMP framework.

1.1.3 Application Servers

A standard way of creating a multi-tier client/server application is using an *application server*, which supplies the mid-tier run-time execution environment and hides away the heterogeneity of the network, providing for smooth integration. Usually the middleware offers an abstraction of the *object bus*, over which the inter-object communication, typically termed *remote method invocations*, are performed.

In addition, application servers take care of the scalability and reliability of the hosted applications. Some of the industry standard application servers, such as *EJB* [21], also tackle the high-availability issues to some extent. However, to the best of our knowledge, no existing application server provides a highly available run-time environment that copes with the kind of failures handled in this work. This issue is elaborated in Section 8.

1.2 HAMSA scope

In light of the above, it should be noted that the decentralization of management comes at a certain price. A distributed management system itself becomes very complex and its behavior becomes so convoluted that even a very experienced network manager would have difficulties handling it. We call this the *meta-management* problem.

Solving the meta-management problem is crucial for the ultimate success of the distributed management paradigm. The administrators should be relieved from the highly non-trivial issues of deployment and control of the distributed management system in presence of various network failures.

We identify a clear requirement for increasing the availability of the critical monitoring components being part of the management applications. Given the complexity of handling distributed multi-tier applications in an unpredictable environment prone to various network failures, it is both important and challenging to provide a maximally transparent infrastructure that allows a manager to deploy the needed monitoring components of the middle tier in a highly available manner. This improves the overall failure behavior of the management applications, enables more efficient applications (such as event-driven monitoring applications), and therefore contributes to better provisioning of network services in general.

In this work we present our novel Highly Available distributed Management System Architecture (HAMSA) and its implementation. HAMSA is an extensible generic platform for the development of distributed management applications with guaranteed behavior in presence of various network failures. Although the current version of HAMSA is primarily oriented towards furnishing the management by delegation approach, it can be generalized to accommodate the mobile management agents as well.

HAMSA supplies efficient system-level solutions for meta-management problems that include support for flexible hierarchical mid-level management structure, built-in high availability and fault tolerance of the delegated management components, check pointing of their activities and load sharing. In order to provide simple, elegant and efficient solutions to these problems, HAMSA utilizes group communication [6], an approach developed for facilitating fault tolerant distributed applications, which has matured over the last ten years. More explanations about the group communication service can be found in Section 3.4.

In this work we present an infrastructure that facilitates highly available application-specific monitoring components and discuss the management applications, which would greatly benefit from exploiting it. Our infrastructure transparently handles complex failure scenarios including network partitions and host failures. The deployment of the infrastructure requires only minor additions to the application code.

2 Model

In this work we assume *general omission model*. This model allows both the network and the host to loose an unlimited amount of messages. This model captures all benign failures including host crashes/recoveries, as well as network splits/merges. More explanations can be found in [14].

In this model, messages may be delayed, lost, reordered, and duplicated by the network; hosts, monitored devices, and network links may crash and recover independently; and *network partitions*, *i.e.*, the independent isles of connectivity, may be formed due to network errors.

We say that two entities reside in different network partitions, if they cannot communicate. We assume network partitions being transient, which means that after some limited time period of γ the disconnected entities will get connected again.

2.1 (k, Δ) -bofo semantics

One of the common approaches for achieving high availability in a client-server system is the *primary/backup* protocol. In the *primary/backup* replication a service is provided by a set of *servers*, while only one server acts as a *primary* at any given time instance. A client initiates requests by sending a message to the server that it believes is the current primary. When the primary server receives the client's request, it processes it, and sends a response. Let us assume, for simplicity, that all client requests are *synchronous*. This means that the client does not initiate a new request until it gets the response to the previous one. Let δ be an upper bound on the time that it takes for a server to process a client's request, and for a message to travel in a round-trip between the client and the server. Then, if a request was issued at time t , and no response is received by the time $t + \delta$, we say that the *service outage* has occurred at time t . If at time $t + \delta + T$ client receives a response to its request, we say that the service outage interval is T .

In the *primary-backup* replication protocol, other servers detect that the current primary became unavailable, and elect the new primary using some algorithm. The new primary continues providing the service from the last service-specific *state* that is known to it. To allow this, primary should propagate the state changes that it makes as a result of processing the client's requests, to other servers. Various *primary-backup* protocols will differ on such issues as when the state changes are propagated, to which other members of the replication group, and on the exact sequence of the operations. The common part among all these protocols is that at some point the primary notifies other members of the replication group about the state change to allow them for transparent take over in case of this primary's failure.

Reference [14] defines the following four properties for the *primary-backup* client/server protocol:

- *Pbl*: There exists predicate $Prmy_s$ on the state of each server comprising the replication group. At any time, there is at most one server s , whose state satisfies $Prmy_s$, *i.e.*, there always is just one active replica of a service in the system.

- Pb2: Each client i maintains a server identity $Dest_i$, such that a service request i interacts only with $Dest_i$. This property, actually, distinguishes the primary-backup replication approach from the active replication, in which the client broadcasts its requests to all the servers in the service replication group. This requires the same order of processing in order to preserve state consistency of all the servers.
- Pb3: If a client's request arrives at a server that is not primary as defined in Pb1, this request is not processed.
- Pb4: There exist fixed values k and Δ such that all service outages can be grouped into at most k intervals of length of at most Δ each. This makes the service to behave like a single (k, Δ) -bofo server. This property rules out implementations, in which the primary ignores all requests from the clients. This property also implies that only a bounded number of failures can be tolerated by the service during its lifetime.

For the *general omission failure* model N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg prove in [14] a lower bound of $n > 2 \cdot f$ on the degree of replication for preserving the consistency of the service state, where f being the maximal number of failures that may occur during a given epoch. This condition basically requires a majority of servers being available to serve clients' requests at any moment, unless in the state of service outage, in order to ensure a single consistent service state.

In this work we also utilize the primary-backup approach. However, we show that different policies can be preferred for communication between the network monitoring services and the monitored devices on one hand, and for communication between the monitoring clients and the services on the other hand. In fact, the general service availability can be significantly improved in terms of the *disconnected* operation during the periods of service outage from the clients' perspective, while still preserving the four primary-backup protocol properties. More details on the proposed algorithms and policies can be found in Section 4.

2.2 Problematic scenarios

In order to better understand the main issues we would have to cope with, if we relaxed the *bofo* properties for a single active replica of a service, let us consider the following service replication scenarios in terms of the general omission model.

Let S denote a network monitoring service responsible for the monitoring of a device D .

Let the two servers, A and B , comprise the service replication group G . Service S is deployed at G , and both servers A and B receive its replica.

Let the state of the service S be defined as a recursive function F_s from the membership of G , in which it has been accumulated and the previous state. The initial state is denoted by \mathfrak{I} . Thus, for example, for the next membership M , the state would be denoted by $F_s(M, \mathfrak{I})$.

1. At the first stage both servers A and B run in the same partition that includes the monitored device D . Let the service S start running on A as its primary server. Having accumulated some initial state, S requests its replication. At this point the state is propagated to B by the primary server A . Therefore the replicated state is denoted $F_s(\{A, B\}, \mathfrak{I})$.

2. Then, due to a network or a host failure, B leaves to a separate partition or simply shuts down, while S keeps running on A and eventually updates its state, which is based on the information from $F_s(\{A, B\}, \mathfrak{J})$. This new state, termed $F_s(\{A\}, F_s(\{A, B\}, \mathfrak{J}))$, cannot be shared with B since it's unavailable now.
3. If A fails now and stops working with D (or shuts down) B may come up and try to recover the service S in its new partition from the most updated state that B is aware of, *i.e.*, $F_s(\{AB\}, \mathfrak{J})$ from (1). This is the state that S had accumulated at A prior to the B 's failure in (2). As soon as B joins the partition containing D , S that is executing on B , starts accumulating a new state, which is unknown to A and is not based on the information from $F_s(\{A\}, F_s(\{AB\}, \mathfrak{J}))$, but rather solely on the information from $F_s(\{AB\}, \mathfrak{J})$. The new state is: $F_s(\{B\}, F_s(\{AB\}, \mathfrak{J}))$.

The above development gives raise for at least three possible scenarios that require a special handling:

- a. In this scenario the service S completes its work and stops running on the server B while the server A is still disconnected. S reports the results of the monitoring task to its manager client. Assume B fails or gets disconnected. Shortly afterwards A comes up and recovers S from the state $F_s(\{A\}, F_s(\{A, B\}, \mathfrak{J}))$, since it is unaware of the fact that S has already completed on B . This may perplex the external clients receiving the information provided by S .
- b. Another option for B is to recognize the failure of A , while A may simply join a different network partition and continue running S in that partition. In such case concurrent external events that may occur at different partitions and involve the same external party may puzzle the latter.
- c. It is also possible that the servers get split in such way that the target device D can be observed from multiple partitions simultaneously, while the servers residing in different partitions do not see each other. Therefore when the servers from the different network partitions are merged together, it is possible that each of them has different states for their respective copies of the same application. Due to concurrency these states might contain overlapping information. This raises the following question: should these states be merged in some way, or should one state be given preference over the other?

It is worth noticing that the first two cases described in subsections (a) and (b), can be considered as a more general one, when two disconnected application's replicas send an output to an external entity in parallel. This is because the work completion (a) can be treated as yet another kind of output, as well as the monitoring observation reports described in (b).

In order to avoid such inconsistencies and preserve the main four properties presented above, a straightforward primary/backup approach would suggest ceasing any activity in partitions that may potentially violate any of the *bofo* properties. However, such approach will probably cause significant loss of the highly valuable monitoring information, while in fact, some of the servers may still have access to the target monitored device and could continue monitoring it while even in disconnected mode.

Therefore, our goal is to find a more suitable approach for the monitoring services replication. The following issues should be handled in order to provide a monitoring service with a consistent state for carrying on its task:

- Obtaining a single consistent service state;
- Preserving as much of the valuable monitoring data as possible;
- Not baffling external entities that interact with the network monitoring application by an inconsistent behavior.

2.3 Problem statement

Having considered the above problematic scenarios we will now define the basic terms and properties of the HAMSA's problem domain.

We use the Lamport timestamp [30] notion of time t in our definitions.

- Monitoring observations consistency: We will use $M_s(t, d)$ to denote a single monitoring *observation* of the monitored device d obtained by the monitoring service S at time t . A monitoring service keeps its *observation history* that is an ordered, possibly infinite, sequence of observations $M_s(t_i, d)$, where the value of t_i serving as the observations' ordinal index.

We say that a service's observation history is *consistent*, if for any two observations $M_s(t_l, d)$ and $M_s(t_m, d)$ for the same device d : $l < m$ if and only if $t_l < t_m$.

- External events consistency: We will call any processing entity, e.g., management client, interacting with a monitoring service an *external entity*. The interactions of a monitoring service with an external entity are of two types:
 - *Input interaction*: the service receives information from an external party;
 - *Output interaction*: the service sends information to an external party;

Any *external event* of interaction between the service S and an external party is denoted as $e_i = In_s(m, p, t_i) \vee Out_s(m, p, t_i)$, where i being the ordinal number of the component external event, m being a message containing the interaction data, and p denoting the external party: the sender in case of the *In* event, and the destination in case of the *Out* event. It is obvious that the events order can significantly influence the service consistency.

We say that two events e_i and e_j are in *causal relationship*: $e_i \preceq e_j$, if the following conditions are preserved ([6] provides more details):

- If $e_i, e_j \in E_x$, where E_x being the external events history of a specific party x , and $e_{x,i} \preceq e_{x,j}$, then $t_i < t_j$.
- If $e_i \in E_x$ and $e_j \in E_y$, where E_x and E_y being the external event histories of the parties x and y respectively, and $e_i = Out(m, x, t_i)$ and $e_j = In(m, y, t_j)$ for the same message m then $t_i < t_j$ and $e_i \preceq e_j$.
- If $e_i \preceq e_n$ and $e_n \preceq e_j$, then $e_i \preceq e_j$.

We will use the term *external events history* for an ordered (possibly infinite) sequence of external events. We say that an external events history is consistent for a specific service, if and only if:

- The external events history on the service side preserves the causal relationship of events;
 - For any external party involved in interactions with the service the order of its interactions with the service equals that of the server-side.
- *Network monitoring service consistency* must be preserved by HAMSA. We say that a service is consistent if and only if:
 - Its monitoring observation history is consistent;
 - Its external events history is consistent.

3 Architecture

HAMSA is a generic flexible architecture that defines the functionality of a highly available run-time environment for multi-tier network monitoring services. Multi-tier monitoring services usually consist of a front-end client GUI at the uppermost tier; monitoring logic that collects and processes management information at the middle tier(s); and finally the monitored device agents at the lower tier. This structuring provides for disconnected operation, meaning that temporary disconnecting of the first tier from the rest of a monitoring service (e.g., due to some failures) does not imply a stoppage of the monitoring activities.

The higher availability of multi-tier management services is achieved by a transparent replication of the services' run-time environment, while the service itself remains virtually unchanged. Since a service implementation is not modified to accommodate to the new execution environment, HAMSA may not assume any specific implementation, and should guarantee that the service's original semantics are not inadvertently destroyed due to the replication.

In order to provide this functionality, HAMSA defines a set of generic interfaces, their semantics, relationships and methods of interaction for the monitoring services deployed at HAMSA. We term such network monitoring services *HAMSA-compatible components*. We use the term *components* (as done in many other middleware software systems) to describe objects that implement a set of the predefined interfaces allowing dynamically to mix and match this object with other objects that conform to the same set of interfaces.

Figure 3 depicts the three-tier structure of HAMSA. Front-end clients from the first tier communicate with the monitoring logic executing at the second tier using some object bus for remote access. HAMSA can be implemented using different middleware technologies for this part of the architecture. Our current implementation uses the standard J2EE technology. Clients locate both the HA-MLMs and the monitoring components through JNDI naming and directory service, and interact with them using the standard *Java RMI* protocol [28].

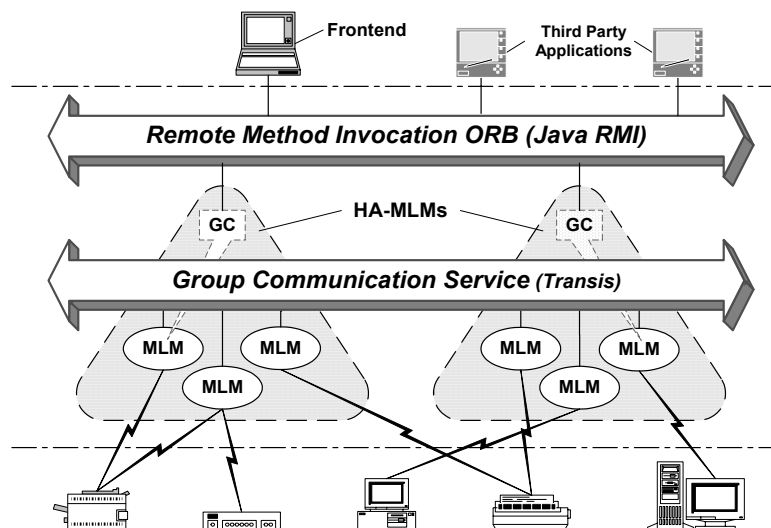


Figure 3 HAMSA Architecture

The run-time environment for the middle tier, *i.e.*, for the components comprising the monitoring logic, is provided in full by the HAMSA framework, as described in the following sections.

The following guarantees on the execution of the hosted network monitoring components in the middle tier generate the primary added value of HAMSA:

- Failures of the components are masked from the outside entities. As long as HAMSA has sufficient resources for executing the components, components function continuously despite host failures and recoveries (*i.e.*, the machine running the component), arbitrary asynchronous network splits and merges.
- In case of network splits, a single instance of each component is executed per network partition.
- The component, whose host machine has failed, is guaranteed to resume operation from the last *consistent* state. We define a state being consistent if no discrepancies exist between its content and any component related data known to the outside world, *e.g.*, management clients.
- Component's interactions with the environment may potentially influence the state of other components, and/or external entities. In this case, interactions (messages, method invocations) are termed *non-idempotent*. Failure, and a subsequent recovery, of a component being in the middle of a non-idempotent interaction may violate the original interaction semantics. An advantage of HAMSA over other middleware architectures is that it preserves the original *at-most-once* or *at-least-once* semantic of the component interactions in spite of asynchronous network failures.

These advantages come at a certain price in bandwidth and processing overhead. In Section 7 we discuss the trade-offs between the extended functionality of HAMSA and this overhead. Also the current HAMSA implementation restricts the inter-process communication model to asynchronous communication and messaging. This communication model, though, fits well into the network monitoring domain.

3.1 Highly Available Mid-Level Managers (HA-MLMs)

The run-time environment with the above properties is provided by a set of virtual servers termed *Highly Available Mid-Level Managers (HA-MLMs)*. HA-MLMs are logical entities that are comprised of one or more physical servers called *Mid-Level Managers (MLMs)*, see Figure 3. To its users, every HA-MLM appears as a single logical entity that exposes the special *HA-MLM Service Interface* (see Section 5 for more details). Each MLM in a given HA-MLM is capable of exposing the HA-MLM Service Interface, but at any given moment only a single primary MLM provides this functionality.

As Figure 3 shows, MLMs communicate with each other using a *Group Communication Service (GCS)* that provides *partitionable membership* semantics, *virtual synchrony* execution model, and *reliable FIFO* message ordering. Section 3.4 provides more details regarding the GCS in HAMSA.

HAMSA encourages the local deployment of an MLM at the managed device wherever feasible, since this will greatly reduce the management traffic overhead and will enable simple solutions for complex problems, such as atomic snapshot of the managed device state.

Network administrators create HA-MLM groups in order to delegate network monitoring components to them later. All MLMs in the managed network join a special "all-MLMs" HA-MLM group termed *HAMSA Enterprise* upon the boot process. New HA-MLMs can be created through this root HA-MLM interface.

The HA-MLMs can be organized into a logical hierarchy as shown in Figure 4. The only constraint of the HA-MLMs hierarchy is that the MLMs comprising the nominal membership of any given HA-MLM must be contained in the membership of the HA-MLMs that are higher in the hierarchy. In other words, if M_A is the set of MLM members of a HA-MLM A , M_B is the set of MLM members of HA-MLM B , and A is higher in the hierarchy than B , then $M_B \subseteq M_A$ must be true.

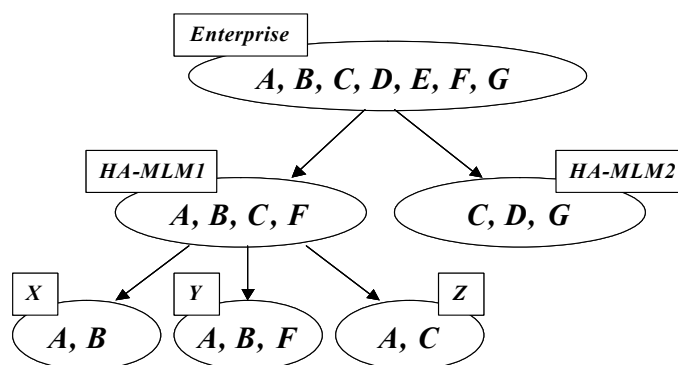


Figure 4: HA-MLMs Hierarchy

3.2 HAMSA-compatible Components

HAMSA provides a special interface for the HAMSA-compatible components' dynamic delegation. The identity of the HA-MLM, to which the component is being delegated, is part of this component's identity.

The MLM serving a specific component delegation request is responsible to propagate the component's meta-data, as well as its executable to all members of the HA-MLM through the group communication service, see Section 3.4. When all MLMs in the HA-MLM receive a copy of a component on their non-volatile storage, the component becomes available for serving external clients.

One of the HA-MLM members is made responsible for running an active replica of the newly delegated component according to some predefined policy. Policies for the primary MLM election can vary based on specific criteria definition, *e.g.*, a load-balancing algorithm. See Section 4.1.6 for more details.

The primary MLM selected to host the active replica of a component is termed *component host*. In the same network partition there can be only a single component host for any given component at any given time. Other MLMs within the same HA-MLM keep dormant replicas serving as warm backups for the components, whose host MLM may fail, see Figure 5. This

is different from the more common practice of keeping components on a component server, and downloading them on demand. HAMSA performs component propagation as above to increase their availability.

The information about the HAMSA monitoring components delegated to a HA-MLM and the HA-MLM's logical hierarchy information comprise the state of this HA-MLM. This state is replicated among the HA-MLM members on a per-update basis using the group communication service. When membership changes occur in the HA-MLM (*i.e.*, MLMs join/leave the HA-MLM group because of failures), members of the group are notified by the group communication service, and they run a state exchange protocol (see Section 4.1.4) in order for the HA-MLM to resume its regular operation.

As a result of network partition changes a HA-MLM can be in either *majority* or *minority* state. This is based on the number of MLM members that are present in the given network partition, compared to the total number of the MLMs that belong to the HA-MLM. The set of all the MLMs comprising a HA-MLM is the HA-MLM's *nominal membership*. In this work we use the term *minority* for any *non-majority* state. See Section 4 for more details on the HA-MLMs behavior in membership changes.

Since the state of a HA-MLM is replicated among all its members, any HAMSA component can be resumed from a consistent point by any other MLM in the HA-MLM that remains operational as explained in the following sections. Each time a component is (re-)activated at an MLM, it also updates the external *naming and directory service* in order to renew the binding between the component's name and its communication handles. Thus, for the network administrator the use of HAMSA creates a reliable environment, in which the delegated tasks "simply do not fail", unless a catastrophic failure occurs.

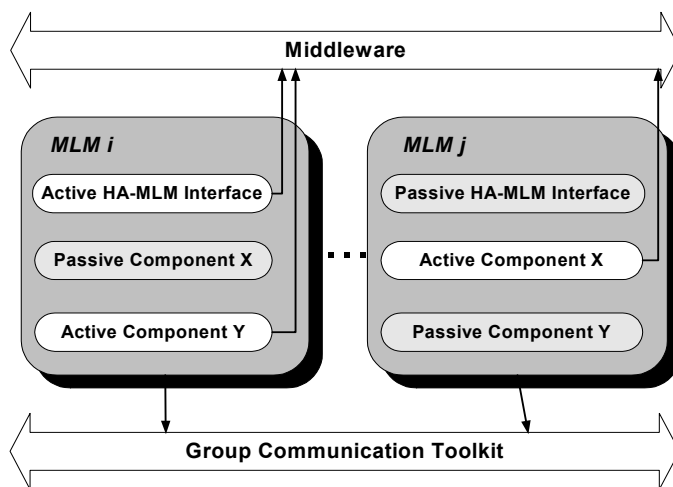


Figure 5: HA-MLM Structure

To render warm backups of the executing components, MLMs transparently replicate the *state* of the components delegated to their HA-MLM. The component state is co-located with the component itself in order to achieve high availability. This is another difference between HAMSA, and more traditional approaches, in which a dedicated database is used to store the state of the components.

The state of a component consists of its *interaction state* and *internal state*. Interaction state contains all the unprocessed inbound and outbound interactions between this component and external entities. It is HAMSAs framework's responsibility to manage the components' interaction states. A component's internal state consists of arbitrary application-specific objects. The internal state objects are managed by the components themselves, while HAMSAs provides the means for reliable backup and recovery of the state objects with no knowledge of their internal semantics. This is done through a dedicated interface that allows components for demanding state replication without knowing any details of the replication mechanism. Assuming that a state replication is required any time it affects a component's consistency, this, along with the additional properties of the HAMSAs algorithms, allows us for preserving the desired HAMSAs components semantics.

Components may interact with other entities executing within the same HA-MLM, in different HA-MLM, and outside HAMSAs, *e.g.*, with the front-ends residing in the first tier, and the network elements.

To provide its guarantees, HAMSAs requires that all non-idempotent interactions between the HAMSAs-compatible components and any external entities are made through *HAMSAs messaging service*, as described in the next section. This allows HAMSAs for transparent replication of the interaction part of the component state.

3.3 HAMSAs Messaging Service

HAMSAs messaging service (HMS) is defined as a generic mechanism providing HA-MLMs' basic messaging services to abstract entities named *recipients*. HMS uniquely identifies a recipient entity by its symbolic name. A recipient name consists of a *principal prefix* that is unique within a specific HA-MLM, and the HA-MLM name. Both HAMSAs components and external parties are treated by HMS as recipient entities. The recipients that are locally hosted by a HA-MLM are termed *local* for this HA-MLM's HMS, all other recipients are termed *remote*.

In order to start getting service from HMS, a recipient entity is supposed to register at HMS of some HA-MLM. The registered recipients enjoy the following set of basic messaging services: *register*, *deregister*, *send* (including HA-MLM level broadcast), *receive*, and *poll*.

We define two basic types of recipients: *synchronous* (capable of receiving messages synchronously via a callback interface) and *asynchronous* (responsible to fetch the pending messages from HMS by themselves).

The HA-MLM assigns two types of communication for each component: *Mailbox*, and *Proxy*. Mailbox is needed for direct sending a message to a component. There is one mailbox per HA-MLM with a separate message queue per a recipient served by this HA-MLM. In order to support remote method invocations, while using the HAMSAs consistency and ordering mechanisms transparently, we use the standard *Java proxy* [28] approach. Any remote invocation between a HAMSAs component and any other party is intercepted, and processed by the per-component proxy. The proxy creates a message from the method call performed on it, and relays it to the HAMSAs messaging service in form of a regular external interaction message. One restriction of this approach is that HAMSAs-compatible component cannot support synchronous method invocations with non-void return values. HAMSAs

communication model requires that if a caller wants to receive information from a component it has to supply either a callback interface or to be registered as a recipient at the mailbox serving this component.

Allowing direct interactions of the components with their environment is not always safe if we wish to comply with the (k, Δ) -*bofo* semantics. HAMSA defines that each component is assigned an interaction approver that policies its interactions. In particular, HMS may defer interactions with the outside entities depending on the specific state of the HA-MLM. In this work we suggest utilizing the majority based interaction approval policy, *i.e.*, a component may send/receive data only when it resides in a partition with majority of the HA-MLM member servers available in the specific network partition.

In order for the HAMSA messaging service to be transparent and straightforward for the parties that participate in the interactions, they should be unaware of the message handling mechanism implementation. For this purpose HMS is responsible to intercept any interaction between a HAMSA component and an external party, and to fulfill it only when the consistency of such interaction and of the whole system can be guaranteed.

However, the interactions between the HAMSA-compatible components and the monitored network elements are not handled by HMS and, therefore, are not restricted in any way, since it is not feasible. This is the key feature distinguishing our solution from the existing approaches [14], and the main reason for HAMSA's being primarily targeted to monitoring, and not to other kinds of management activities that may affect the state of the target devices.

As described above, interactions with the target devices are supposed to be implicitly reflected in the component-specific state objects, which get replicated either on demand from a component that owns it, or transparently to the component, upon an outgoing interaction initiated by the component.

An important property of HMS is its distinguishing between incoming and outgoing events for communication with external parties. When a HAMSA component receives an event from external entities, it does not enforce an immediate update of the replicated state among the HA-MLM members hosting this component. This is because incoming events do not immediately affect the component consistency, as defined in Section 2.3. However, it is not the case with the outgoing events. Therefore, upon an external interaction attempted by the component itself, the replicated state of the component is synchronized among all the MLMs prior to handling the outgoing message. As we explained in Section 2.3 this leads to the following key property of HAMSA:

- *A HAMSA component that potentially made its internal state publicly available through an interaction with external entity would never return to a more outdated state in spite of the errors found in the general omission failure model.*

3.4 Group Communication Service (GCS)

Group communication service is a message passing service that provides a concept of a *process group*. Each process group is a group of end-points communicating according to the many-to-many model and referred to by a single logical name. A message sent to the group is delivered according to the same requested order to every member of the group (or to nobody),

provided that the member does not crash. A group communication service is usually comprised of a *membership service* and a reliable *message ordering service*.

The group membership is a list of group members that are considered connected and active, and that the group members agree on. The objective of the membership service is to handle various asynchronous events, such as: network partitions and merges, host failure and recovery, group members join and leave. The membership service notifies the group members about the membership changes, so that any two members that belong to the same membership are indeed connected.

The replication of the components state within a HA-MLM is facilitated by a group communication toolkit that is not visible outside HA-MLM (see Figure 3). Such toolkit systems usually allow processes for forming groups that can be addressed by a single logical name, so that messages can be sent to the group using this name as an address, and all operational members of the group receive them. HA-MLMs are realized in HAMSAs as process groups.

HAMSAs rely on the GCS toolkit to provide the following capabilities:

- *Reliable multicast FIFO* delivery of messages.
- Per-group notification of membership changes either due to network failures, or members (i.e., MLMs in the context of HAMSAs) voluntarily joining/leaving the group.
- *Virtual synchrony model* of message delivery, which, simply stated, means that members of the group that go together through the same set of membership changes, receive the same set of messages.
- *Partitionable membership model*, which means that although members of the same group can find themselves in different network partitions (due to asynchronous network splits), each connected component can continue its operation. And when a network merge occurs, the members can resume operation from a consistent point in the message stream so that the virtual synchrony model is preserved.

There are a number of group communication toolkits available that supply this functionality, such as [7]. The HAMSAs architecture does not dictate any specific choice of GCS. It is also possible that as the middleware technology advances, the GCS with the needed semantics will become an integral part of a standard object middleware, therefore removing the need for an external GCS.

4 HAMSA algorithms

In order to facilitate higher availability of a management component without changing the components themselves, HAMSA has to deal with the failure possibilities, giving rise to multiple conflicting states of a component transparently. The primary requirement from HAMSA with respect to the failure handling is to conform to the failure model and external events semantics expected by the management component that is not aware of being replicated. It is also desirable to minimize the loss of the valuable information, when a common consistent state is obtained from the multiple states of the component's replicas.

HAMSA allows different network partitions to make progress concurrently. It automatically handles the consistency maintenance issues arising from the complex failure scenarios. As described in Section 2, restarting a component from a warm backup in case of a failure raises the following non-trivial problems:

- a. A management component may depend on its state checkpoint. Such components are termed *history-dependent*. When a history-dependent component is recovered at a different MLM, this should be done from a consistent point with respect to the component's state and to the interactions with external parties that could have taken place prior to the component's recovery.
- b. It is possible for several replicas of the same component to execute simultaneously. On the other hand, external parties treat the component as a single logical entity and are unaware of the replication mechanism. Enabling external entities to access different replicas of a component inconsistently would severely baffle an external party communicating with the component, and would violate the (k, Δ) -*bofo* server semantics, as defined in Section 2.1.

Concerning the history-dependent components, the design of HAMSA was made along with the following guidelines. First, the only thing that a component is aware of is, that in case of a failure at its current host, it will be restarted at some other MLM. Second, the component should not get involved with the details of its state replication. Third, the only responsibility of a history-dependent component is to maintain its state as an arbitrary object or a set of objects, and inform the hosting MLM about the changes it makes to its state.

In case of a history-independent component, it is straightforward to resume it at a different location, since this only requires deciding, which MLM should take care of it. This decision can be easily achieved thanks to the membership notifications from the GCS received by all the HA-MLM members at times of the network and host failures. Having the membership notifications provided, there is no need in additional agreement protocol on the MLM's level, since these notifications are already the output of a distributed agreement protocol that is run at the GCS level.

In regard to obtaining a common state out of multiple conflicting states, one should observe that there is actually no way to merge multiple states, due to the possibility of the concurrently executing component instances. In this case, some parts of the component executions may overlap, and, if consistency is the issue, only one of the conflicting states can and should be preferred.

We will show that, in fact, we have fairly good means to resolve these conflicts in a meaningful way. We would prefer the state of the component instance that has more external communication events (completed, or being in progress) with an involved external party. The intuition behind this rule is that the more communication with the external party has been performed, the more information is in the component’s copy state. What is even more important, this rule allows us for satisfying one of the main consistency requirements defined in Section 2.3, namely, preserving the execution semantics, as expected by external parties.

4.1 Algorithms description

The following sections present the major HAMSAs algorithms. These algorithms constitute the main logic of the key HAMSAs mid-tier element, the MLM. HAMSAs MLM’s logic is event-driven. It is based on two basic event types:

- *Membership change*: is an event provided by the group communication service that informs about a HA-MLM membership change. Section 4.1.2 presents the HAMSAs algorithm for handling this type of events.
- *Regular*: is either an internal HAMSAs event provided through the group communication service from some other HAMSAs mid-tier entity, or an external interaction event issued via remote method invocation by an external party. Section 4.1.3 presents the HAMSAs’s external events handling algorithm.

In addition, the HAMSAs state exchange, load balancing, and garbage collection algorithms are discussed in the following sections.

For the sake of simplicity we present the HAMSAs algorithms assuming that there exists only one monitored device. We also assume that the executables of the HAMSAs components have already been successfully propagated and initialized at all the MLMs in the HA-MLM’s nominal membership; and the initial responsibilities for the components execution have been assigned among the MLMs.

4.1.1 HAMSAs states

As described in Section 2.3, HAMSAs deals with two main stateful entities: HA-MLM and HAMSAs component. Table 1 and Table 2 describe the variables comprising these entities’ states respectively. We will use these states definitions in the algorithms’ pseudo-code in the following sections.

Variable	Description
<i>N</i>	Nominal MLMs membership of the HA-MLM
<i>Components</i>	Meta-data of the components that are currently hosted by the HA-MLM
<i>Mailbox</i>	Per-component interaction state, <i>i.e.</i> , buffer of pending messages,
<i>Progress</i>	HA-MLM progress reflects the number of external requests resulting in a HA-MLM update, see Section 4.1.4

Table 1: HA-MLM state

Variable	Description
S	Internal component state based on the monitoring observations history obtained by the component copy executing on a specific MLM host
<i>Progress</i>	HAMSA Component progress reflects the number of outgoing events issues by the specific component, see Section 4.1.4

Table 2: HAMSA component state

4.1.2 HA-MLM membership change handling

A HA-MLM membership change may result in one of the following changes:

- An MLM in a majority partition of a HA-MLM can move either to (1) a majority partition with different configuration or to (2) a minority partition.
- An MLM in a minority partition of a HA-MLM can moves either to (1) a majority partition, or to (2) a minority partition with different configuration.

In case of a membership change with a merge of partitions containing multiple replicas of the same component with different accumulated states, we need to ensure that a single component replica is executed with a single consistent component state. This is done through the HAMSA *state exchange protocol*, as explained in Section 4.1.4.

Figure 6 presents the algorithm used by HAMSA to handle the membership changes.

Let V and V' be the previous and the new membership of the handled HA-MLM. These variables will also be used in the following sections.

```

    If  $|V'| \leq \lfloor N / 2 \rfloor$  then {                               // We moved into a minority partition
        Purge HA-MLM's Mailbox
        IneractApproval := false
    } else                                                     // We moved into a majority partition
        InteractApproval := true
    If not  $V' \subseteq V$  then {
        Perform HA-MLM and components state exchange // see Section 4.1.4
        For each component and the HA-MLM itself do {
            Elect the host MLM based on the load balancing policy
            If the local MLM is elected then
                Activate the entity (component/HA-MLM)
            Else If an entity is active at this MLM then
                Move the entity into the dormant state
        }
    }

```

Figure 6: Membership change handling

We elaborate the load-balancing policies mentioned in this algorithm in Section 4.1.6.

4.1.3 External event handling

The requirement for external interaction events handling is to obligate all components to perform any such interaction only through the special HAMSAs messaging service mechanism presented in Section 3.3. For the sake of high availability the HAMSAs messaging mechanism propagates any event to all the available MLMs comprising a specific HA-MLM. This mechanism should defer and accumulate any event, either sent by a component or to be received by a component, until the consistency conditions are fulfilled.

We say that the external interactions are permitted, if and only if the majority of the HA-MLM's members is present in the specific HA-MLM membership. I.e., any interaction with an external application is restricted and will be deferred as long as only half or less than half of the MLMs comprising the HA-MLM are present. Once the condition is met, all the delayed interactions should be “released” and fulfilled by HAMSAs. See the algorithm presented in Figure 7 for more details.

```
Let e be the handled event
If InteractApproval == false then {
    Defer e
    Finish
}
If e.type is outgoing external interaction
    Replicate the state of the component e.sender to all the HA-MLM members
    Deliver e to e.recipient
```

Figure 7: External events handling

4.1.4 State Exchange

The HAMSAs *state exchange* protocol specifies the mechanism for obtaining the most updated state of a HAMSAs entity in case of a HA-MLM membership change. The HAMSAs state exchange protocol follows the state machine replication approach. It is designed to treat both HA-MLMs and HAMSAs components as generic stateful objects. Therefore, HAMSAs utilizes the same protocol for exchanging states of all the HAMSAs entities.

As discussed in Section 4.1.2, one of the main issues that the HAMSAs state exchange protocol faces is obtaining a single consistent state out of potentially multiple state instances coming from different partitions. The straightforward approach is to select a single whole state as is from one of the merged partitions. Another option is trying to merge multiple state instances into a single state on behalf of the component. Unfortunately, there is no common mechanism to merge component-specific state without intervening into the component specific internal implementation. Thus, we need to pick the “right” state instance. For this purpose we introduce the *state progress* parameter to indicate how updated a specific state instance is.

In order to ensure the component state consistency it should be replicated on any external interaction request. However, since only outgoing interaction can expose component state

related data to external entities, the state replication can be omitted in case of incoming interactions. We use the number of times a state was replicated, or the number of outgoing external interactions, as the key parameter of the HAMSAs entity state's progress, since this is the main indicator of a potential state change.

In fact, we probably might find a more updated state instance in terms of the accumulated monitoring information; however, we in such case we would violate the component's consistency.

In this section we describe the state exchange protocol of a single HA-MLM. However, in order to reduce the recovery time overhead in case of a HA-MLM membership change, HAMSAs allows multiple HA-MLMs for executing their state exchange protocols in parallel without interfering each other.

The HA-MLM state exchange protocol consists of the following two basic phases marked with the white color in Figure 8:

- *State advertisement*: all the MLMs hosts distribute the state object descriptions on behalf of the hosted stateful objects to other HA-MLM members. For each state instance, an MLM concludes, which HA-MLM member should be the state object's *update source*, based on the progress parameter values received from other MLMs;
- *State distribution*: every MLM checks, whether it is the update source of some of the hosted states instances. If there are such objects, the MLM distributes them to all the members of the HA-MLM through the group communication service.

In order to make the state exchange as efficient as possible our protocol utilizes a single state advertisement message containing the progress details of all the state objects taking part in the state exchange process. Nevertheless, the distributed state objects are sent in separate messages due to their potentially large size. Every message should reach all the MLMs in the membership, therefore, each state exchange protocol requires $n \cdot (n + m)$ messages, where n representing the number of state advertisement messages that is equal to the membership size of the HA-MLM, and m representing the number of the state distribution messages, which in turn is equal to the number of the state objects hosted by the specific HA-MLM.

The state machine diagram in Figure 8 depicts the main steps of the protocol.

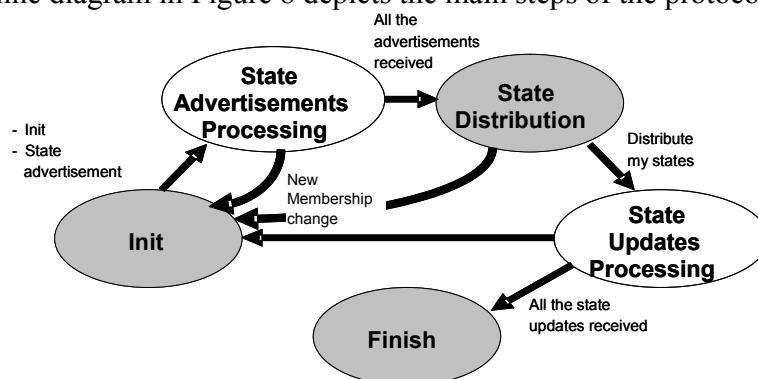


Figure 8: HAMSAs State Exchange protocol's state machine

Data structures:

- *StateSource*: { *sourceName*, *stateDescriptor* }

State exchange initialization:

- *Objs* := \emptyset // a set of stateful objects participating in the state exchange
- *Fellows* := *V* // a copy of current membership set
- *Fellows.remove(MyMLMName)*
- Add the HA-MLM to *Objs*
- Add states of all the components that are hosted by the HA-MLM to *Objs*
- *Sources* := a map of *StateSource* objects of size |*Objs*|
- For (*i* = 0; *i* < |*Objs*|; *i*++) do { // init *Sources* with my MLM for all objs
 Sources.put(MyMLMName, Objs[i].getStateDescriptor())
 Objs[i].suspend()
}

State Advertizement:

- *MyDescriptors* := \emptyset
- For all entries in *Objs* do: add *Objs[i].StateDescriptor* to *MyDescriptors*
- Multicast *MyDescriptors* to the HA-MLM members

Wait for all available MLMs' state advertisements until the *Fellows* set gets empty.

When a state advertisement message (*HisDescriptors*) arrives, do {

- *Fellows.remove(HisDescriptors.getMLMName())*
- For all entries in *HisDescriptors*:
 If (*HisDescriptors[i].progress* > *MyDescriptor[i].progress*) or
 (*HisDescriptors[i].progress* == *MyDescriptor[i].progress* and
 HisDescriptors.getMLMName() < *MyMLMName()*)
 Sources.put(HisDescriptors.getMLMName(), HisDescriptor)

}

State Distribution:

- For (*i* = 0; *i* < |*Sources*|; *i*++)
 If *Sources[i].getSourceName()* == *MyMLMName* then {
 Distribute *Objs[i]* // I am the source
 Sources[i].remove()
 }

Wait for all the state updates. When a state message (*HisState*) arrives, do {

- *Objs[HisState.getName()].update(HisState)*
- *Objs[HisState.getName()].resume()*
- *Sources[i].remove()*
- If *Sources.isEmpty()* then: Finish

Figure 9: HAMSAs state exchange protocol

In case an additional state exchange is required due to a new membership change, while another state exchange process is already in progress, the currently executing state exchange protocol is immediately reset and restarted. The pseudo-code in Figure 9 describes the HAMSA state exchange process in detail.

4.1.5 Garbage Collection

HAMSA messaging service propagates any external interaction message to all the HA-MLM members upon its arrival. As described in Section 4.1.3, HAMSA delivers these messages to their recipients only if the external interactions are permitted.

For the sake of efficiency HAMSA replicates the component internal states only at *outgoing* external interactions, while the propagated *incoming* messages are stored in the MLMs memory as part of the HA-MLM state. In case of a failure, storing the incoming messages in the MLMs' memory allows for avoiding potential loss of valuable information. This is achieved by enabling retransmission of these messages to a recipient component, when it is restarted at a different MLM location.

When an outgoing external interaction is dispatched by a component, the HAMSA policy requires replication of the component's internal state, and therefore, ensures up to date consistent synchronization of the dormant replicas. According to the policy we propose, this is the most appropriate time for the HAMSA messaging service to perform the garbage collection procedure by purging the stored incoming messages.

Additional policies can be considered, depending on the specific requirements of the message delivery service. For example, in case maximum consistency is required, and no message losses can be tolerated, it is possible to synchronize the component state on every external interaction regardless its direction. More relaxed limited mailbox capacity, watermark-based policies can also be considered.

4.1.6 Load-balancing

As explained in Section 3.2 HAMSA implements a special policy in order to elect a primary MLM for running an active replica of a specific HAMSA component.

The simplest type of such policy is the *preferred location* one, which applies, for example, for finding the most proximate available MLM for a specific monitored device. This, in fact, static scheme also implies implicit enforcement of a load-balancing policy by spreading the monitoring tasks load among different MLMs, assuming that the monitored devices are spread as well.

In addition, dynamic load-balancing policies can be applied to the components' load distribution decision. The research of the dynamic load-balancing policies is out of scope of the current work.

4.2 Primary/backup protocol discussion

It was shown in [14] that the general omission failure model implies the lower bound of $n > 2 \cdot f$ on the degree of replication in the primary-backup protocols, where n is the number of replicas, and f is the number of failures.

Now, let us consider HAMSA's algorithms in light of this model. First, we observe that concerning the interactions between the external clients and the HAMSA components (and among the different components in the second tier), HA-MLMs that host these component behave like a (k, Δ) -*bofo* server with respect to every component. Indeed, for each component there is a single primary MLM in the HA-MLM group at any given time, providing the component with the communication services. If a majority of MLMs fails in HA-MLM, and the interaction approval policy is majority-based, no communication with the component is possible. However, in HAMSA this does not imply the service outage. In fact, components continue running, and perform monitoring on the target network elements. It is just the results of their activity that are temporarily unavailable.

We relax the single primary server property when it comes to communication between the components and their target network elements. In fact, we allow multiple primaries operating in different partitions simultaneously, and gathering the monitoring information concerning the network elements in their respective partitions, with one primary instance per partition. However, to preserve the *bofo* server semantics to the higher-level clients of these components, we allow at most one primary server (the one that executes in the majority partition, if exists) to communicate with entities other than the monitored network elements.

To appreciate HAMSA model, consider a non-replicated monitoring application that communicates with some manager at one hand, and with the network elements on the other hand. If this application is stateful, then it should be capable of storing its state on the non-volatile storage each time it considers that the loss of the information gathered thus far is undesirable. If such an application fails, it is subsequently re-started. Upon a restart this application re-reads its state from the non-volatile storage, and continues its operation from the last consistent point that it exposed to the outside world. In HAMSA, we preserve this single object image of the application, but at the same time we shorten the service outage time (*i.e.*, the impossibility to continue the monitoring process) by consistently replicating the application, so that the monitoring continues even if less than $2 \cdot f$ replicas exist in a given network partition.

Thus, in our three-tier architecture, communication between the management clients (the uppermost tier entities), and the servers indeed obeys the restrictions imposed by the low bound on the primary-backup protocols. We allow the periods of unavailability of the mid-level entities performing continuous monitoring tasks to the management clients, when such communication may compromise a single object behavior of the monitoring component. However, for communication of the monitoring components with the target management agents (the bottom tier), we relax the demand of [14], on having a single primary system-wide at any given time instance, since the read monitoring operations are *idempotent*.

We allow multiple primaries to operate simultaneously in non-communicating network partitions transparently to the high-level clients who still regard them as a single entity. Upon a re-merge of the previously split network, our middleware automatically obtains the most updated monitoring state about the disconnected network elements that have been available in some network partition, and were unreachable in the others. To the best of our knowledge, HAMSA is the only monitoring middleware with such high-availability guarantees, specifically designed to address the meta-management challenges of the three-tier monitoring architecture.

5 Implementation Highlights

5.1 Overview

We implemented a full HAMSAs prototype and the initial performance results are presented in this work.

The current implementation of HAMSAs uses Java programming language [28] and therefore is platform independent. We tested it on Windows 2000/XP and Linux operating systems.

We use an advanced group communication toolkit called Transis [7] that provides a virtually synchronous execution model [12] and transparent handling of network partitions and merges, as required in Section 3.4. This toolkit allows us for ensuring the fault tolerance of the core architecture components.

We utilize standard *J2SE* implementation of *Java Naming and Directory Service (JNDI)* for locating HAMSAs entities, and *Java Remote Method Invocation (RMI)* [28] for accessing the HAMSAs distributed system by external entities.

We invested much effort in making HAMSAs easy to install, configure, and use. This is achieved by the provided *Installation and Administration Guides* (see the Appendices). In addition, HAMSAs software is built of independent modules with pluggable internals for potential further research, customization and extension, as explained in the following sections.

The HAMSAs implementation consists of two main parts:

- The HAMSAs MLM server that implements the core HAMSAs logic and provides the main functionality of the HA-MLMs mid-tier framework;
- The administration GUI tool providing a human administrator with a user-friendly graphic interface to HAMSAs.

The following sections present the current HAMSAs implementation highlights.

5.2 HAMSAs mid-tier framework

5.2.1 MLM

HAMSAs MLM server provides an execution environment for the hosted HAMSAs elements. The main element an MLM is responsible for is the logical HA-MLM replication group entities. According to the policies described in Section 3.1, an MLM may belong to a number of different HA-MLMs.

MLM is implemented as a server-side daemon that should execute at any host comprising the HAMSAs framework.

MLM maintains a set of the HA-MLM groups it belongs to. Each element of this set is an instance of the object implementing the HA-MLM functionality, as described in Section 5.2.4. Only one MLM becomes the primary for each HA-MLM according to its predefined

location policy. The HA-MLM instance hosted by the elected primary MLM becomes active and registers itself at the naming and directory service to enable other parties' access.

Being an execution environment provider, an MLM is responsible to provide a number of services necessary for normal execution of hosted elements, and for those elements to conform to the HAMSA semantics, as defined in this work.

The following sections describe the main services provided by MLM.

a. Group Communication service

The group communication service utilized by HAMSA is provided by an underlying group communication service toolkit. Specific implementation of the GCS toolkit is out of scope of this work. MLMs integrate with the GCS toolkit to provide the group communication services to the hosted HAMSA elements. The current implementation relies on the Transis GCS toolkit [7] for reliable group communications services and the messages ordering guarantees defined in Section 3.4.

MLM provide the hosted elements with the following group communication services:

- *HA-MLM membership discovery* allows for obtaining a list of the MLMs currently comprising the membership of a specific HA-MLM.
- *Listen to HA-MLM membership change events* (see Section 4.1) in the HA-MLMs, which the MLM belongs to. The MLM receives asynchronous network partitioning and merging events from the GCS toolkit, and passes them for handling to the relevant HA-MLM.
- *Listen to regular events* (see Section 4.1) related to the elements hosted by the MLM. Upon a regular event arrival the MLM is also responsible to route the received message to the appropriate HA-MLM.
- *Multicast event* to entities in both, locally hosted and external HA-MLM groups;

b. State Persistence service

MLM provides its hosted elements with a persistent check-pointing mechanism for better recovery in case of crashes and/or network partitioning. This mechanism is responsible to maintain the state objects of the hosted elements on a non-volatile storage through the following services:

- *Commit* a state object of a hosted element by writing a copy of the object instance onto a non-volatile storage, *e.g.*, a file system.
- *Reload* a hosted element's state object from its non-volatile storage.
- *Decommission* a hosted element's state object that is no more in use.

c. Naming and Directory service

As described in Section 3 HAMSA uses a *naming and directory service* to enable locating of the HAMSA entities by an interested party. Similarly to the group communication service the actual implementation of the naming and directory service is out of the HAMSA's scope. Since the whole HAMSA prototype was implemented in Java, we utilize the standard JNDI [28] service.

Therefore, the MLM server provides its hosted elements with a set of services listed below through a straightforward integration with the standard JNDI service implementation.

- *(Re-)bind* an object instance to its name.
- *Lookup* a reference to the currently bound object instance by its name;
- *Unbind* the current object binding by its name.

d. HAMSAs Execution Environment Sandbox

The components executing at a specific MLM server are provided with virtual sandbox mechanism that enables isolation of the HAMSAs components from each other and from the underlying system implementation.

The current sandbox implementation is very basic. It relies on the standard Java built-in mechanisms, such as *class loader* and *security manager*. For more details refer to the standard Java 2 SDK documentation [28].

5.2.2 HAMSAs component

HAMSAs compatible component is a standard Java archive (*JAR*) package containing at least one class implementing the component interface specified by HAMSAs. A component provider must define the main class of the package to be used by HAMSAs. A HAMSAs compatible component is launched by calling its main class's *run* method. The components are executed in separate threads of the hosting MLM's *Java Virtual Machine (JVM)*.

We support a basic component versioning mechanism that can be extended in the future. The current implementation ensures that only the most updated component version will be executing.

In order to increase its usability, HAMSAs provides a set of easy-to-use tools for adjusting a ready application to a HAMSAs-compatible component, as well as for developing of new HAMSAs-compatible network monitoring components. In order to make the development of the HAMSAs components easier we suggest the following two basic approaches:

- In case you already have a ready network monitoring application with its own complex logic, and it is a regular Java application, you may prefer utilizing the automatic HAMSAs wrapper utility that will automatically adjust your application to be compliant with HAMSAs requirements;
- In case you prefer to enjoy more of the HAMSAs features, you can utilize the provided basic implementation of HAMSAs component. If you use the class extension technique to build your monitoring logic on top of the basic component implementation, you will save the need to implement the default behavior of HAMSAs component, such as its naming interface, state maintenance, and basic runtime statistics management.

One can also refer to the provided sample reference component implementation that utilizes the HAMSAs basic component implementation.

5.2.3 HAMSAs stateful object and state

HAMSA provides a generic definition of *stateful object*. Both HA-MLM and HAMSA component are implemented as stateful objects.

Stateful object is any entity that is interested in keeping its state persistent in form of a set of HAMSA *state objects* that comply to the specified implementation guidelines and interfaces. State object is an arbitrary object treated by HAMSA as a “black box”. The only requirement for HAMSA compatible state object is to provide its identification and progress. The progress parameter is used by the state exchange protocol to identify the most advanced state instance.

One could implement the state progress as any comparable metric, but, in order to meet all the HAMSA guarantees, the suggested progress metric, namely the number of outgoing external interactions, must be used. Nevertheless, in case weaker guarantees are sufficient for a specific component or its specific state object, a different progress metric can be preferred.

HAMSA identifies the state objects by their name, which must be unique within the scope of a specific stateful object. HAMSA maintains the set of multiple state objects belonging to the same stateful object in the structure named *object state map*. A state map contains all the state objects of the specific stateful object (0 or more), and provides an access to them by their names.

5.2.4 HA-MLM

As presented in Section 3.1, HA-MLM is an entity providing generic hosting, replication, messaging and execution mechanisms for network monitoring services, *i.e.*, for HAMSA components. A HA-MLM entity maintains all the data required to manage the HAMSA’s flexible hierarchical framework for execution of HAMSA components.

HA-MLM entities are hosted by the MLM servers that belong to the membership of the specific HA-MLM. For each HA-MLM, a member MLM is elected to serve as the *primary* one similarly to the way the primary MLM is selected for running HAMSA components. The chosen MLM is responsible to register the HA-MLM instance at the naming and directory service, so that other parties would locate and enjoy this HA-MLM’s.

HA-MLM’s implementation consists of three layers providing the three main HA-MLM roles, namely: (1) group hierarchy manager, (2) messaging service provider, and (3) HAMSA components hosting and execution framework. Each one of these layers relies on the functionality provided by a lower layer, as described in the following sections.

In addition, certain parts of the HA-MLM implementation are implemented as easily replaceable plug-ins, as shown in Figure 10. For more detailed plug-ins description see Section 5.2.5.

a. HA-MLM group hierarchy management

The HA-MLM groups have hierarchical tree structure. A user may create a sub-group of any HA-MLM node in the tree using one or more MLMs comprising this HA-MLM, as explained in Section 3.1. By default there is one general HA-MLM group composed of

all the registered MLMs. An MLM is preconfigured to automatically join this group by its default name, e.g., “enterprise”, upon initialization.

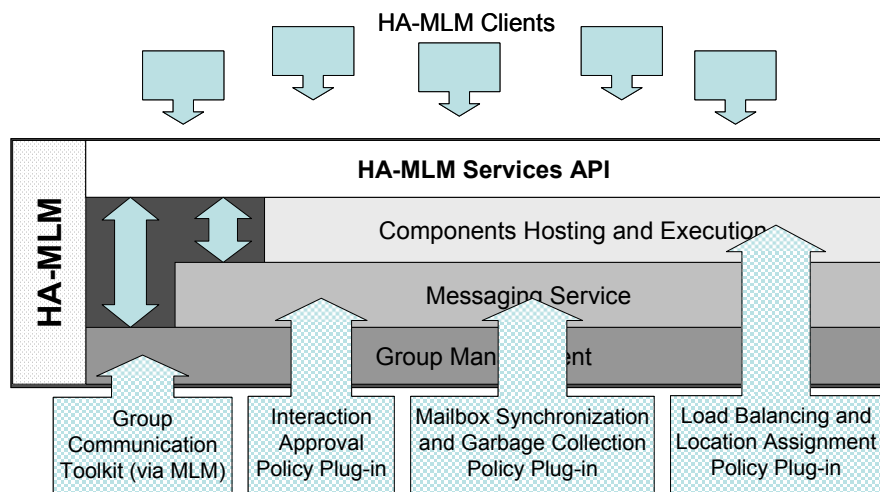


Figure 10: HA-MLM internal structure

The HA-MLM group hierarchy manager is the lowest layer of the HA-MLM implementation. It utilizes the group communication service provided by its MLM server for receiving indications of network partitioning resulting in the HA-MLM’s membership changes.

As stated in Section 4.1, a HA-MLM is capable of handling two basic event types: *membership change* and *regular*. Therefore, our implementation provides two handlers responsible for handling of these events that are routed by the MLM to the specific HA-MLM instance.

The membership change events handler utilizes the algorithm presented in Section 4.1.2. In case a state exchange is required, it also initiates the state exchange process described in Section 4.1.4.

The regular events handler follows the policy defined in Section 4.1.3, and activates an appropriate sub-handler responsible for the specific type of handled event. The regular events can be either of internal type related to some system activity, or of external type containing an external interaction message. The former type’s sub-handlers are provided by one of the three HA-MLM layers according to the internal event sub-type, while the latter type is handled by the messaging service, as described in the next section.

The HA-MLM group hierarchy manager serves two basic types of consumers: (1) HAMSAs administration clients responsible to administer and maintain the HAMSAs framework, and (2) hosted HAMSAs components. These services enable the *tree* hierarchy management.

The following services are currently provided by this HA-MLM layer:

- *Add/remove sub-group HA-MLM*: creates a new HA-MLM under the current one in the HA-MLMs hierarchy;

- *Get parent HA-MLM group*: Since the only supported hierarchy structure is the tree, there always is only one parent for any given HA-MLM except for the root one that is termed *enterprise* and has no parent node;
- *Get a specific sub-group HA-MLM* by its name;
- *Get all sub-group HA-MLMs*;
- *Get nominal membership*: supplies a list of the MLMs comprising the nominal HA-MLM membership;
- *Get currently effective membership*: supplies a list of the MLMs comprising the HA-MLM membership in the current network partition. It must be a subset of the HA-MLM's nominal membership;
- *Get a historical membership view*: for sake of future statistical investigation, an HA-MLM also keeps the historical data regarding its past memberships.
- *Get name of the MLM server* currently hosting the specified HA-MLM;
- *Get/set the HA-MLM status (active/suspended)*: a HA-MLM is automatically suspended when it resides in a network partition with minority (non-majority) of its MLMs available;

b. Messaging service

The messaging service is the middle layer of the HA-MLM implementation. On top of the group hierarchy management it provides the HA-MLM's highly available messaging functionality.

The messaging service layer handles the generic *recipient* entities and provides all the messaging-related services they need, as described in Section 3.3. It also maintains a *mailbox* with separate queues of messages for all the recipients. A HA-MLM uses its mailbox to store pending messages, in case their immediate delivery is not possible for some reason. In case of an outgoing external event the messaging service is responsible to initiate the mailbox synchronization and garbage collection process, as explained in Section 0.

In addition, the messaging service layer of HA-MLM is responsible for the following activities:

- *Register/deregister recipient*: a HAMSAs recipient must be registered at one of the HA-MLMs in order to enjoy the messaging services. At the registration time a recipient is provided with a unique name comprised of the recipient name prefix part and the HA-MLM name using the standard URI [29] email-like notation, *e.g.*, *aaa@bbb*, where *aaa* being the recipient name prefix, and *bbb* being the name of the HA-MLM the recipient is registered at. In order to incorporate HAMSAs into more generic framework of networking services in the future, the HAMSAs specific URI service prefix can be added to the HAMSAs recipient name structure, *e.g.*, *hamsa:aaa@bbb*.
- *Send message*: a message can either be sent to a specific HAMSAs recipient by its full name, or broadcasted to all the recipients registered at a specific HA-MLM by using the asterisk as a recipient name prefix, *e.g.*, **@bbb*.

Message sending is an asynchronous process. Normal return status of the *send* service call only means that the message has been successfully propagated to all the HA-MLM members as an outgoing event. It does not guarantee the immediate message delivery.

- *Receive message*: once called this service blocks until a new message is received for the recipient that called it. If the message is already available in the recipient's mailbox queue, it returns immediately.
- *Poll message*: similar to *receive*, but it does not block. *Poll* returns *null* in case no messages are available for the recipient that called it.

Figure 11 depicts how a message is communicated to a recipient component. (1), The sent message is not delivered to the target component immediately. Instead, (2), it is propagated to all MLMs in the HA-MLM using the group communication service. When, the message arrives at the group communication service level at all operational MLMs, including the propagator, (3), the MLM host routes the message to the relevant HA-MLM that puts it into the component's mailbox queue. Then, (4), the messaging service consults the component's interaction approver. If the interaction approver permits the interaction, (5), the message is delivered to the active target component. See Section 4.1.3 for the algorithm's description.

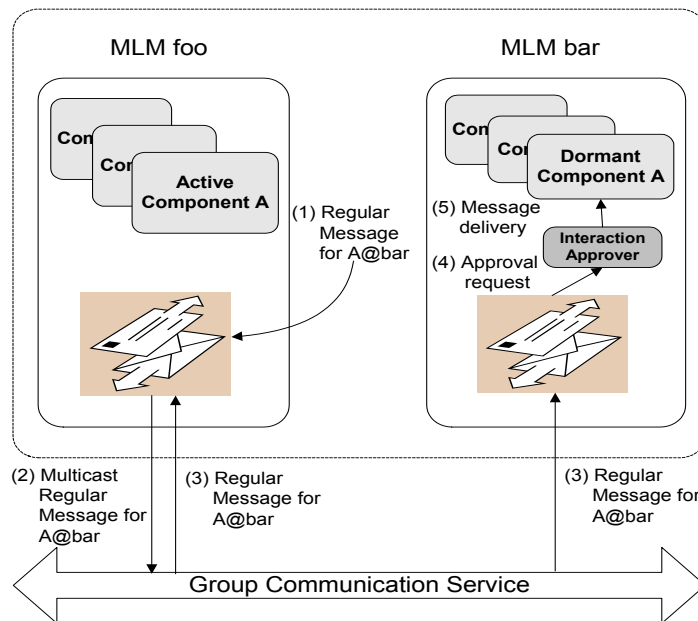


Figure 11: HAMSA Messaging Service

c. HAMSA components hosting and execution service

The components hosting, execution, and life-cycle management service is the top layer of the HA-MLM implementation. It is the only layer that is aware of the network monitoring components existence, and, therefore, provides all the services required for the HAMSA components delegation, replication, and execution:

- Everyday component lifecycle services, such as *sandbox* functionality provided through the MLM hosting this HA-MLM.
- Fault tolerance related services, such as components state storage and replication management provided through the MLM, state exchange in case of a network partitioning.
- *Add/remove component*: This is the HA-MLM component delegation/removal service. Adding a new component includes its delegation, replication to all the HA-MLM members.
- *Start/stop component*: An administration client can request a HA-MLM to launch, suspend, and resume a specific component.
- *Get components*: provides a set of components interfaces for the components that are hosted by the HA-MLM.
- *Get status*: indication whether a component is running, suspended, deactivated, etc.
- *Get state*: allows for obtaining the current component's state map.

5.2.5 HAMSAs plug-ins

As shown in Figure 10 the current implementation of HAMSAs supplies the following plug-ins that can be easily replaced for testing of different algorithms with HAMSAs framework:

- *Group communication toolkit*: is provided through the described above MLM interface;
- *Interaction approval policy*: is used by the messaging service. It implements the logic of the decision whether a specific interaction is approved or not. The current HAMSAs prototype provides an implementation for the majority-based interaction approval policy, as described in Section 3.3;
- *Mailbox synchronization and garbage collection policy*: provides the messaging service with the functionality described in Section 0. The current HAMSAs prototype utilizes the synchronization and garbage collection mechanism for purging pending messages at every state replication caused by an outgoing external interaction.
- *Load balancing and location assignment policy*: assists the components hosting and execution service to identify the most appropriate location for a HAMSAs component's execution, as explained in Section 4.1.6.

5.3 HAMSAs administration tool

The HAMSAs administration tool is a graphic front-end interface that allows a human administrator for performing various HAMSAs administration and maintenance tasks. This tool is provided as an example of HAMSAs framework management client implementation. Using the provided APIs it is also possible to develop other management clients.

Our administration tool was implemented with strong focus on its usability. It has a user-friendly file explorer-like graphic interface with the HA-MLM hierarchy tree on the left side

and the display area providing the information related to the selected HA-MLM on its right side, see Figure 12.

The administration tool enables working with three basic entities: *HA-MLM*, *messenger*, and *HAMSA component*. HA-MLM and HAMSA component entities were described earlier. Messenger is a graphic representation of the HAMSA messaging service recipient entity. It connects to the messaging service of a HA-MLM as its recipient, and allows for sending and receiving text messages to any other HAMSA recipient, including other messengers and HAMSA components.

The left-side HA-MLMs tree of the graphic interface enables the navigation through the potentially complex HA-MLMs hierarchy. HAMSA enterprise HA-MLM is the root node by default. In addition, all the HA-MLM nodes are marked with an appropriate color to indicate their current status.

- Green color means that the HA-MLM is reachable and ready to serve any request.
- Yellow color means that the HA-MLM is reachable, but can only serve “read” requests, since it is suspended, *e.g.*, due to the interaction approval policy constraints, *i.e.*, the current membership of the HA-MLM does not contain majority of the MLMs.
- Red color means that the HA-MLM is unreachable.

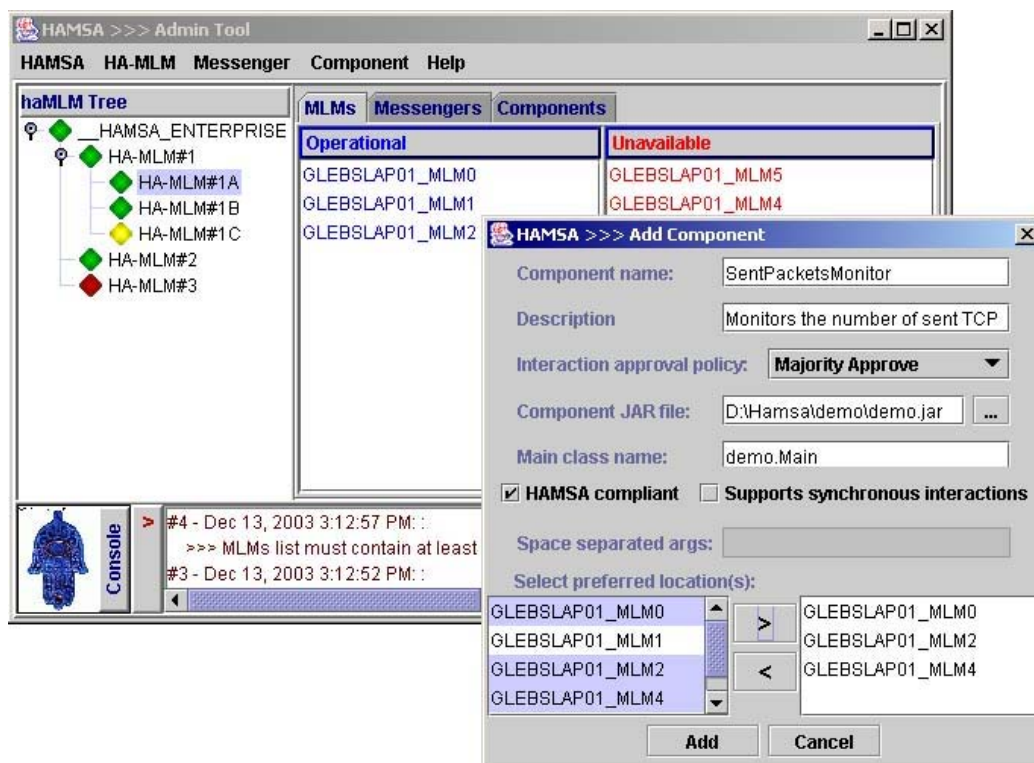


Figure 12: HAMSA administration tool

Having selected a reachable HA-MLM in the left-side tree, one can choose any of the provided three HA-MLM information views on the right side, namely: (1) the current and the

nominal MLMs membership, (2) the HAMSAs messengers registered with this HA-MLM, (3) the HAMSAs components hosted by the HA-MLM.

The top part of the application contains the menu-driven toolbar that allows for performing various actions on the described above entities. And, finally, the bottom part displays the HAMSAs console that allows for keeping track of possible error, warning and notification messages.

The following functionality is provided through the toolbar menus:

- *File menu:*
 - *Refresh:* resets the connection of the administration tool to the HA-MLMs by reconnecting to the root HAMSAs enterprise HA-MLM;
 - *Exit:* closes the application;
- *HA-MLM menu:*
 - *Add:* opens a dialog for the new HA-MLM creation;
 - *Remove:* removes the selected HA-MLM;
- *Messenger menu:*
 - *Add:* opens a dialog for the new messenger creation. A new window is opened for each newly created messenger;
 - *Reopen:* reopens the window of already existing messenger. Prior to this action the messenger must be selected in the *Messengers* view on the right side of the application window. Instead of using this action from the menu, one can get the same effect by double-clicking on the selected messenger.
 - *Remove:* removes the selected messenger;
- *Component menu:*
 - *Add:* opens a dialog for entering the data necessary for the new HAMSAs component creation. The mandatory fields are:
 - *Component name:* the specified name may not contain spaces and/or special characters. This name will be used as recipient name prefix, while the component's full name will be comprised of the provided name and the HA-MLM name.
 - *Interaction approval policy:* specifies the policy to be used for the specific component.
 - *Component JAR file:* specifies the location of the JAR file containing the component logic.
 - *Main class name:* specifies the name of the class implementing the HAMSAs component interface. The name should include the full package name.
 - *HAMSAs compliant:* if not checked, the administration tool assumes that the added component does not implement the HAMSAs component interface, but rather contains a regular Java application *main* method.

In such case the automatic HAMSAs component wrapper is activated to wrap the specified application with default HAMSAs interface implementation.

In addition one can specify the component's description and a space separated list of arguments that the application *main* method will be invoked with. The latter option is relevant only for non-HAMSAs compliant components.

- *Start*: in case the selected component has not been started yet, it is initiated and started. If the component was suspended, it is resumed. Otherwise, it does nothing.
- *Stop*: suspends the selected component's execution. Please notice that both start and stop actions implementation rely on the JVM-specific thread suspension and resuming implementation.
- *Remove*: removes the selected HAMSAs component;
- *Help*:
 - *About*: provides some general information about HAMSAs.

5.4 Implementation structure

The current HAMSAs prototype is implemented in *Java* language, and therefore, its modules are grouped into packages according to their functionality. The following list summarizes the structure and content of the main HAMSAs packages in alphabetical order. We also provide a very brief description of the most important classes in these packages.

5.4.1 *Admin* package

This package contains the HAMSAs administration tool implementation. *AdminGui* is the main class of this application.

5.4.2 *Component* package

This package contains the HAMSAs component related functionality:

- *ComponentIntf* specifies the interface an application should implement in order to become a HAMSAs compatible component;
- *BasicComponentImpl* provides the default implementation of the basic HAMSAs component functionality that can be extended with some application specific logic. This module is intended to save developers' efforts for developing new HAMSAs components.
- *ComponentState* is used internally by HAMSAs to maintain the component's meta-data, *i.e.*, the data provided at the component delegation. *ComponentState* object is replicated similarly to other HAMSAs state objects as a part of the HA-MLM state. *ComponentState* uses *ComponentVersion* as its progress metric;
- *ComponentStruct* is used internally by HAMSAs to maintain the component's runtime data. This structure is kept for active component replicas only;

- *ComponentVersion* implements the component versioning logic;
- *PerHostComponentStatistics* maintains the statistical data regarding a component's activation history on different MLM hosts. It is utilized as one of the default state objects in the *BasicComponentImplementation*'s state map;

5.4.3 Core package

The *core* package contains the core modules of the HAMSAs functionality, namely:

- *MLM* is an implementation of the MLM server functionality, as described in Section 5.2;
- *HamsaEvent* is the basic HAMSAs event definition;
- *HamsaInteractionApprover* is the majority based interaction approval policy implementation;
- *Host2GroupIntf* specifies the interfaces provided by an MLM to its hosted HA-MLM objects.

5.4.4 Exceptions package

This package contains the exception classes' definitions used by HAMSAs.

5.4.5 Group package

This package implements the group hierarchy management functionality of HA-MLM. It also serves as a relay for the messaging service layer implemented in the *HamsaMailbox* module of the *messaging* package. *I.e.*, for the messaging service we use class composition rather than direct inheritance.

- *Group* is the main class of this package implementing the core logic of HA-MLM;
- *GroupIntf* specifies the HA-MLM's subset of interfaces that are related to the group hierarchy management functionality;
- *GroupState* and *GroupProgressMetric* implement the HA-MLM's meta-data state and its progress metric respectively;
- *GroupStateXchangeProtocol* extends the basic *StateXchangeProtocol* implemented in the *state* package. It provides the HA-MLM group specific functionality required for the full state exchange of a HA-MLM;

5.4.6 HaMLM package

This package extends the *group* package's modules with the messaging service to provide the HAMSAs components hosting and execution layer of HA-MLM:

- *HaMLM* is the main class of this package. It extends the *Group* class and implements the core components hosting and execution logic of HA-MLM;

- *HaMLMIntf*, *HaMLMState* and *HaMLMStateXchangeProtocol* extend the appropriate classes of the *group* package to complete the HA-MLM's the components related functionality;

5.4.7 *Logger* package

This package provides the HAMSAs proprietary generic severity-based logging mechanism with extensible model of multiple logger agents.

- *Logger* implements the main logging logic implemented as a singleton class. It allows the logger agents for subscribing to logging events of specific severities. Its main *log* method handles all the logging events and passes them to the relevant subscribed logger agents;
- *Severity* specifies the set of severity supported by the logging mechanism. The currently defined severities are: *debug*, *GUI alert*, *notification*, *warning*, *error*, and *fatal error*;
- *LoggerIntf* specifies the interface to be implemented by a logger agent;
- *DefaultLogger* provides the basic implementation of a logger agent;
- The rest of the classes in this package provide specific implementations of various logger agents, e.g., *FileLogger*, *PrintStreamLogger*, etc.;

5.4.8 *Messaging* package

This package contains all the modules involved in the HA-MLM messaging service:

- *MessageServiceIntf* defines the API of the HA-MLM messaging service;
- *MessageClientIntf* defines the interface that a message service recipient should implement;
- *HamsaMailbox* implements the HA-MLM's messaging service layer. It is incorporated into the *Group* module implementation;
- *HamsaMailboxPolicy* specifies the mailbox synchronization and garbage collection policy;
- *HamsaMessage* specifies the basic message entity used by the messaging service. This class extends the *HamsaEvent* one that is defined in the *core* package;
- *HamsaRecipient* implements the recipient specific part of the *HamsaMailbox*. It maintains the recipient's queue of pending messages;
- *ExternalInteractionMsg* extends the *HamsaMessage* to represent a HAMSAs external interaction event;
- Rest of the classes in this package implement various message types that are internally used by HAMSAs;

5.4.9 *Plugins* package

This package contains two types of modules: those implementing not HAMSA-specific generic functionality, and HAMSA plug-in modules. The latter, such as *FileHelper* and *JNDIHelper* utilities, generic FIFO queue, alarm and monitor implementations, are used by most of the HAMSA modules in all packages, while the usage of the latter was describes in Section 5.2.4.

5.4.10 *State* package

This package contains all the modules dealing with the HAMSA state management, replication, and exchange. Section 5.2.3 provides more details on the functionality of the following modules:

- *StatefulObjIntf* and *BasicStateFulObjImpl* provide the interface and the default implementation of the entities that are interested in HAMSA's state replication services, *e.g.*, HA-MLM and component modules implement this interface;
- *StateHostIntf* specifies the interface that a state replication service provider should implement. HAMSA *MLM* module implements this interface;
- *StateMap* implements the entity maintaining a set of state objects belonging to the same stateful object;
- *StateIntf* and *State* provide the interface and the default implementation for component specific state objects;
- *StateDescriptorIntf* and *StateDescriptor* specify the interface and the default implementation of the state descriptor that is a parameter provided by the state object implementer. It contains the state progress metric.
- *StateXchangeProtocol* provides a generic basic implementation of simultaneous state exchange process for multiple stateful objects. It follows the algorithm specified in Section 4.1.4;

5.4.11 *Transis* package

The *Transis* package provides the interface to the Transis group communication toolkit. Its specification can be found in [7].

See the Appendices for the additional information on how to use HAMSA

The following sections provide analysis of the current HAMSA model behavior and overheads.

6 HAMSA component applications

In this section we present typical network monitoring applications, demonstrate how one can deploy them using HAMSA, and explain the benefits network monitoring applications gain from taking the HAMSA approach.

6.1 Post-mortem failure analysis

The first network monitoring application is a highly available post-mortem failure analysis system. In this application, several MIB scalar variables from each network element are being kept in a centralized repository, and when a network failure occurs, the management system searches this repository for the relevant variables, whose values may suggest the source for the failure (see for example [22]).

In a typical two-tier scenario such a system is deployed at a single station, and the MIB variables of all network elements are accessed from it. The collected data is kept in the local file system. When a failure of a monitored element (or of several elements from the same network region) is detected the collected data is searched and the behavior of the relevant MIB variables is examined in order to identify the cause of the problem.

In HAMSA, the centralized polling application and its repository are being handled transparently by the middleware. The administrator chooses a set of MLMs by either selecting an existing HA-MLM, or defining a new one, and delegates the polling component to this HA-MLM. Based on the component placement policy, the controller activates this polling component at one of the MLMs, while the replicas are kept for warm backup at other MLMs.

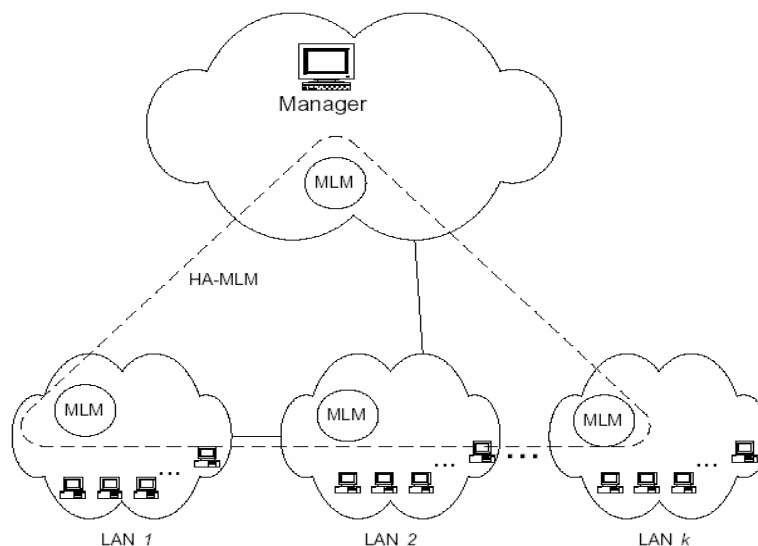


Figure 13: All MLMs are put into a single HA-MLM

If the network splits, the monitoring continues automatically in each network partition where at least one MLM of the split HA-MLM is present. The state (e.g., the collected MIB variables), is kept locally per replica of the monitoring component in each network partition.

When the network re-merges these autonomously collected states become available to the administrator.

One question raised by this example concerns different configuration trade-offs available for the monitoring application that uses HAMSA. Consider the typical network configuration illustrated in Figure 13. In this scenario, the information arrives at the monitoring station from k LANs. If the monitoring is done by centralized polling from the management station, and the connectivity to one of the LANs is lost, the monitoring of its elements cannot continue. In particular, if the failure is caused by a misconfigured access interface in the LAN's access router, the information about the cause of the problem will not be available. This is because the connectivity may be lost before the values of the router's MIB variables suggesting the cause of the problem are retrieved.

If however the administrator configures HA-MLM in such a way that there is at least one MLM per LAN, the MLM in the disconnected LAN will re-start a separate copy of the monitoring as soon as it detects (through the underlying group communication service) that there is a network partition, and all variables in the router's MIBs will be polled.

Once connectivity is re-established (say, through rolling back the configuration) the management station will be able to access this information, and the manager will be able to identify the source of the problem, (i.e., wrong configuration) and to fix it.

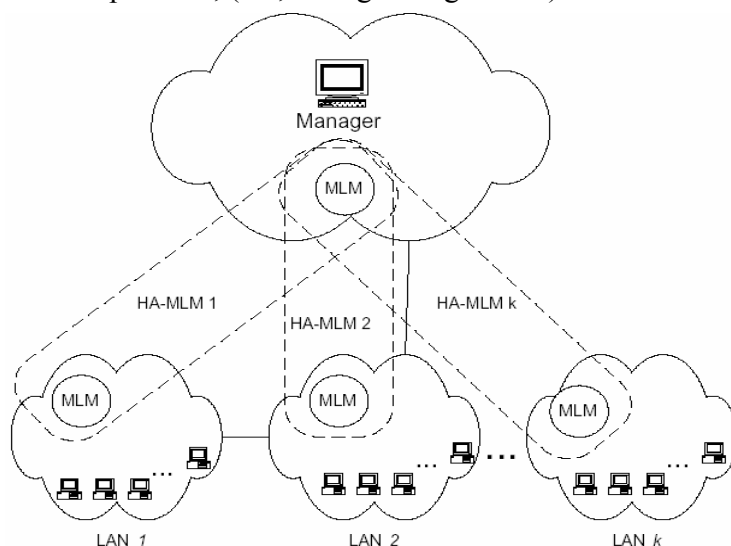


Figure 14: Pair-wise Organization

This example also demonstrates the importance of proper HA-MLM configuration. The administrator may be tempted to have at least one MLM in each LAN, as in our example. However, since the state of each monitoring component is distributed by HAMSA to all members of the HA-MLM, the communication costs induced by the replication may become too high.

In fact, one may choose to create m separate applications, each having a different HA-MLM containing only a pair of MLMs, as described in Figure 14. In this case, the monitoring application for each LAN is running separately on the local MLM (according to the distance-based component placement policy), and thus being unaffected by a possible network

partition. If, however, the local MLM itself fails, a copy of the monitoring process for that LAN will be initiated automatically by HAMSAs on the MLM that is co-located with the management station. This configuration also reduces the overall monitoring traffic when there are no failures, since in this case the monitoring is done locally and the state of the monitoring components is synchronized among the two MLMs only upon the external interactions.

There exists a trade-off between the monitoring overhead traffic, and the overhead traffic induced by HAMSAs due to replication it performs behind the scene. The actual amount of overhead depends on the total number of MLMs in a HA-MLM, the size of the application state in HAMSAs, the frequency of external interactions, and the amount of data involved in these interactions.

For example, in the described post-mortem failure analysis application, one can choose to have a small state (i.e., the serial number of the last poll), or a very large state (i.e., the actual data of the last 10 minutes polling). Clearly, the latter choice allows a faster recovery after a failure of a monitoring component, but it generates much more overhead traffic. We study these trade-offs in Section 7, and show that the overhead required by HAMSAs to provide the extra functionality is much smaller than the monitoring costs we saved.

6.2 Event-driven reactive monitoring

A more complex monitoring application demonstrating the inter-process communication capabilities of HAMSAs is an event-driven reactive monitoring network monitoring application. In such an application we are required to detect when a function (typically the sum) of a number of MIB variables, each belonging to a different network element, exceeds a predefined threshold.

A centralized realization of this application involves a polling station that monitors all variables at all network elements, computes the function and sets up an alarm if the value has exceeded the threshold. This solution induces both significant traffic overhead and computation load at the monitoring station that grow linearly with the number of polled elements.

To address these issues, several algorithms that combine local computation, traps, and a centralized monitoring station were proposed in [10]. However, in order to deploy these algorithms the agent should be able to carry on simple computation components and issue traps, which are in many cases beyond the ability of the standard SNMP trap framework. This is a very good example where the extended functionality of HAMSAs can be utilized. The global reactive monitoring application is executed in HAMSAs in a distributed way. Namely, a number of copies of the same monitoring component are launched at several HA-MLMs. Each HA-MLM is responsible for its own local set of devices. According to the algorithm of [10], if a local threshold event has been detected (in this case the “local” means “with respect to the local set of variables”), then the other copies of the monitoring component are being notified using HAMSAs messaging service. Then according to the algorithm, a global poll may be initiated, and, if needed, an alarm is declared. If one of the local monitoring processes fails, HA-MLM controller restarts it on another MLM, and the system continues functioning. This, of course, comes with a cost of increasing the monitoring

traffic, but paying such a cost is definitely better than losing the ability to carry on with the critical monitoring component.

6.3 Usage-based IP billing

As an additional example, consider the following possible organization of a usage-based IP billing application that relies on its own network monitoring components to collect the needed information. The structure of this application is shown in Figure 15. In the lowest tier of the application there are target agents capable of providing the raw information about their respective devices. In the second tier there are three types of monitoring components: flow collector, session collector, and flow preprocessor. The flow collector component monitors the forwarding devices and obtains the raw data about the IP flows from them directly.

The session collector component monitors some Authentication, Authorization and Accounting server(s), such as RADIUS [9], collecting the user-session information. The flow preprocessor component monitors the flow collector and the session collector in order to correlate the users with their flows and reports this pre-processed information to the upper tier. The upper tier of the application consists of the billing component that generates the per-user bills.

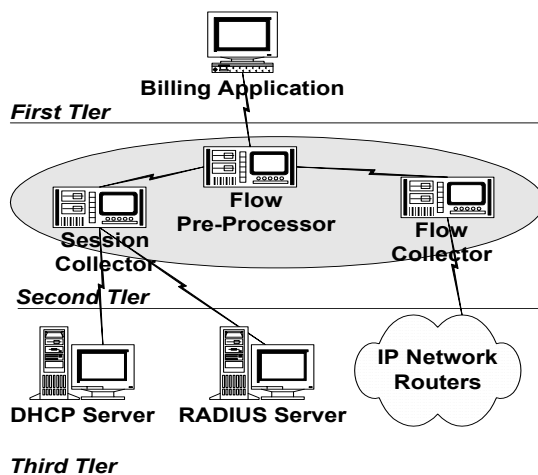


Figure 15: Usage-based IP billing application

Although a failure of any component at the second tier does not imply the immediate stoppage of all the application activities, the billing application, as a whole, cannot proceed in a regular manner and eventually would halt or become inaccurate if no additional measures are taken. For example, if the flow pre-processor component fails then at some point the flow collector will run out of the storage space and will be forced either to lose valuable information, or to seize its activity.

In light of the above we identify a clear requirement for increasing the availability of the critical monitoring components being part of the management applications. Providing a maximally transparent infrastructure that allows a manager to deploy the needed monitoring components of the second tier in a highly available manner would improve the overall failure behavior of the described applications, and therefore contribute to the better provisioning of the network services in general.

7 Performance evaluation

7.1 Trade-off analysis

In order to understand the trade-off between the communication overhead induced by HAMSA, and the possible reduction in monitoring overhead, consider again the scenario described in Figure 13, and Figure 14. We want to compare the amount of traffic overhead generated by the monitoring application without HAMSA with the overhead induced by HAMSA and the underlying group communication service.

The group communication service is responsible for failure detection that is based on periodic broadcasting of short *I-am-alive* messages. In general, this overhead grows as m^2 , where m being the size of HA-MLM. Optimizations that reduce by factor l , the number of LANs, are possible [23]. However, this is inevitable overhead of failure detection that cannot be strictly attributed to HAMSA or group communication, because any application wishing to achieve the high availability guarantees of HAMSA on its own would pay these costs anyway. The experiments performed in [23] with the current implementation of Xpand and Transis show that group communication scales to 200 hosts dispersed over WAN without visible impact on the regular traffic.

The overhead of HAMSA itself strongly depends on the way we configure HA-MLMs and on the size of the application *state*. In order to investigate the trade-off, assume that the state size sent by HAMSA is 150 bytes. This is a reasonable size, when one chooses to use a small state (like a measurement sequence number).

Figure 16 depicts the tradeoff for the two choices of HA-MLM configuration and for 10, and 20 scalar MIBs variables in each LAN. We assumed here that due to the SNMP encoding, polling of one variable takes about 150 bytes, and thus polling 10 or 20 variables per LAN will consume 1500, and 3000 bytes respectively for each LAN.

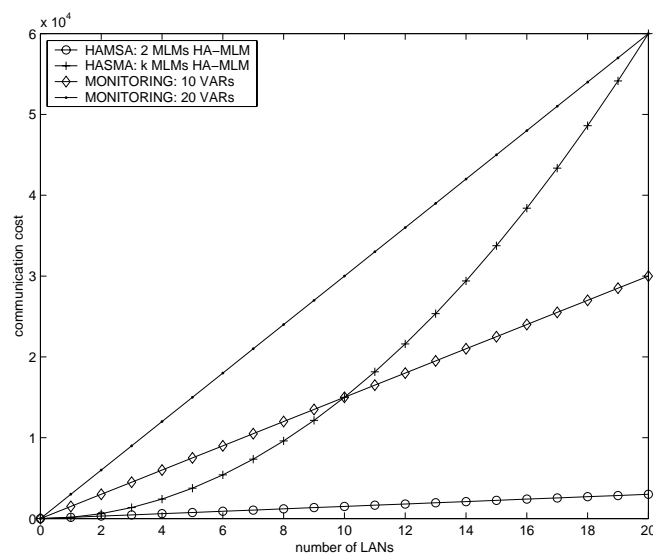


Figure 16: HAMSA and monitoring communication cost as a function of the LANs number

Let n be the HA-MLM membership size, and m be the number of hosted monitoring component with a single state object per component. Assume we need to update all m component states each polling interval. In this case the state replication takes $150 \cdot n \cdot m$ bytes. On the other hand, if we use m different HA-MLMs, each of size 2, HAMSAs overhead is reduced to $300 \cdot m$.

One can easily see that even for a very small number of monitored variables the overhead of HAMSAs is significantly smaller than the monitoring overhead of a traditional application. This is a big advantage even without considering the HAMSAs main goals: extended functionality and reliability.

The main mission of HAMSAs is increasing the system reliability. However, the high reliability comes with the cost of introducing more MLMs. In particular, this implies a higher communication overhead. Thus, there is a trade-off between the level of availability and the traffic overhead. In order to evaluate this trade-off, we consider the same scenario as above.

The current host MLM of an active component replica propagates its state to the rest of its HA-MLM through multicast. Thus, communication cost of HA-MLM replicating the state is linear in the number of group members. However, when we increase the number of MLMs in the group, we reduce the probability of a total system failure, since HA-MLM restarts the failed process on a different MLM as long as they are available.

Thus, if we have m MLMs in a HA-MLM, and the independent probability of a single MLM failure is p , the probability of the application failure is $\tilde{p} = 1 - p^m$. Since we have only one active component per network partition then the communication is $s \cdot (m - 1)$ per each state in the component state, where s being the component state size. To obtain specific numbers, let s be 150 bytes, as in our example. Then, in order to get an application failure probability of \tilde{p} we pay $150 \cdot (\lceil \frac{\log(1-\tilde{p})}{\log p} \rceil - 1)$ bytes per change. This cost is plotted in Figure 17, for single MLM failure probability of 0.1, 0.01, 0.001, and 0.0001.

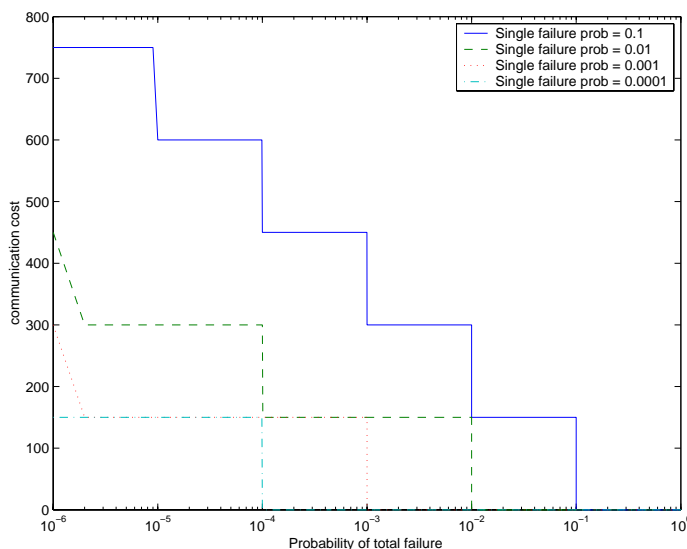


Figure 17: HAMSAs communication cost per state change as a function of the required system error failure probability

As one can see, in order to get the often desired “5 nines” reliability, starting from a very high error rate of 0.1 on a single machine, 6 MLMs are sufficient. The communication cost $150 \cdot (6 - 1) = 750$ bytes per state change) becomes much smaller when the reliability of a single machine increases.

7.2 Experiments

In our experiments we evaluated the following HAMSA capabilities:

- *Messaging throughput*: We measured the HAMSA’s overhead in the messaging interactions between an external client and a hosted HAMSA component;
- *MLM host recovery overhead* as a function of:
 - HA-MLM size;
 - Number of components;
 - Component state size;

The following sections describe the achieved results.

7.2.1 Test-bed setup

In our HAMSA experiments we used the following configuration:

- Hardware:
 - *Servers*: Intel PIII 800MHz with 512MB RAM;
 - *Network*: 100Mbps Ethernet LAN;
- Software:
 - *Java SDK*: 1.3.1;
 - *OS*: Debian Linux 3.0;
 - *Group Communication Service toolkit*: Transis [7] utilizing group multicast;

We used a set of HAMSA components with different state sizes that implemented the “echo” messaging functionality, *i.e.*, the components sent an automatic reply to the sender of every received message. Our components were implemented based on the provided HAMSA component infrastructure that utilized all the HAMSA framework capabilities. In particular, the components’ states were replicated and saved to the file based persistent storage at each outgoing reply message.

7.2.2 Messaging throughput

In this experiment we evaluated the state replication overhead applied by HAMSA in case of a simple request-response interaction between a management client and a monitoring component that requires state replication. We measured the time passed from the moment a client issued a request message till the reply arrival. The main goal of this experiment was to see whether the state replication overhead significantly affects the communication throughput.

Therefore we executed the same test case with different state replication frequency. Since the state replication in HAMSAs takes place only at an outgoing interaction time, we modeled a component that ignores a preconfigured number of requests, replying only on every x -th message. *I.e.*, the higher the value of the parameter x is, the lower the frequency of the state replication is.

We expected the total overhead to get lower as the number of required replications was reduced. In other words, the less we interact with a component, the less we pay in synchronization overhead. We used the following values for the replying *i.e.*, replicating state, frequency: every 1st, 2nd, 4th, 8th, 16th, 32th, and 64th message, where 1 being the highest frequency and 64 being the lowest one. Despite our straightforward and reasonable estimation, Figure 18 shows that for the values we used the HAMSAs's state replication overhead is minor.

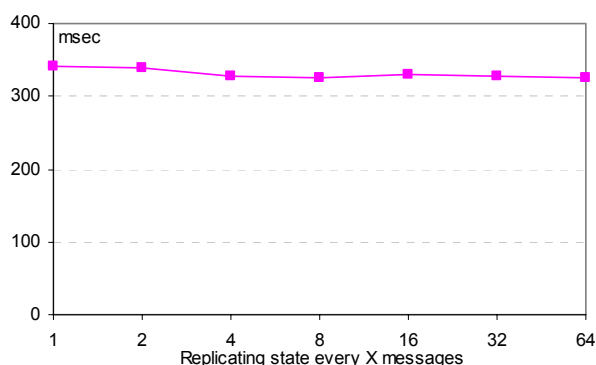


Figure 18: Request-response roundtrip time as a function of state replication frequency

7.2.3 MLM recovery overhead

This set of experiments intended to evaluate the HAMSAs's overhead in case of network/host failures causing HA-MLM membership changes.

In each test scenario described below we set up a system configuration of two HA-MLMs, *i.e.*, we created a dedicated HA-MLM to host the components in addition to the default *HAMSAs Enterprise* one. In our experiments we analyzed the behavior of the MLMs in the dedicated HA-MLM. We used test components maintaining two internal state objects each. These states represented the application internal logic checkpoints and the accumulated monitoring data respectively. The former one had a constant size of 1K. We changed the size of the latter according to the test case requirements. All the components had the same preferred location MLM assignment policy, *i.e.*, all the components always had the same primary MLM server.

We defined a set of major scenarios to test the recovery capabilities of the HAMSAs MLMs. Each scenario focused on the influence of a specific HA-MLM parameter on the MLM recovery overhead. We tested how the following three parameters affect the MLM's recovery overhead in terms of time and number of messages:

- *Membership size of the HA-MLM*: three different sizes: 3, 5, and 7 were tested. In this scenario the HA-MLM hosted a single component with two state objects of 1K and 10K respectively. See Figure 19;
- *Number of hosted components*: we used similar components with two state objects of 1K and 10K each. The number of components varied as follows: 0, 1, 5, 10, 20. See Figure 20;
- *State size of the hosted component*: The HA-MLM hosted a single component with two state objects, one of 1K, the other one with different sizes: 0K (no second state), 1K, 10K, and 100K. See Figure 21;

In all the experiments we analyzed the behavior of each one of the MLMs in the replication group. Let us consider a HA-MLM H with at least three MLMs. Let A , B , and C be MLMs in H . Assume all the MLMs available in the same partition in the beginning. Let us analyze the possible scenarios that an MLM is supposed to cope with in case of a membership change.

1. Let the current partition of H contain all the MLMs except for A ; and let B be the current components' primary server. Then let MLM A join this partition after being either down or in some other network partition. Assume A is supposed to become the primary server for the components according to their preferred location policy. In this case we measure:
 - a. The effort required for A to initialize itself, to synchronize its state objects with other MLMs, and to take over the responsibility for running the monitoring components;
 - b. The effort required for B and C (B is the state update source) to participate in the state exchange process, while the responsibility for hosting the active components moves to A .
2. Assume now that A is leaving the common partition due to its host or network link failure. Then one of the remaining MLMs (in our scenario it is B) should take over the tasks that A was responsible for. In this case we measured:
 - a. The effort required for B to take over the activities of A . Please notice that this case differs from the case (a) in the previous scenario, since no initialization and state exchange is required this time;
 - b. The effort required for C to handle the membership change event. No state exchange is required in this case according to the algorithm in Section 4.1.4.

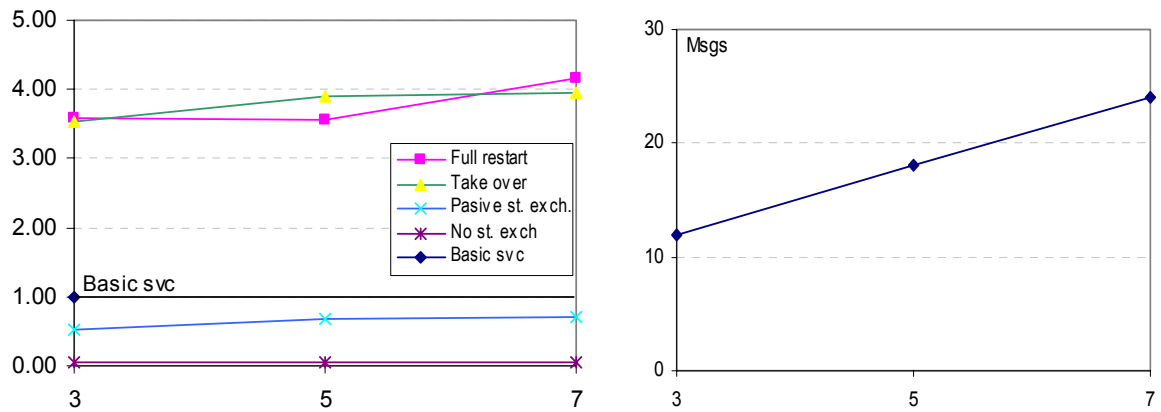


Figure 19: a) MLM recovery time; b) number of messages as a function of HA-MLM membership size

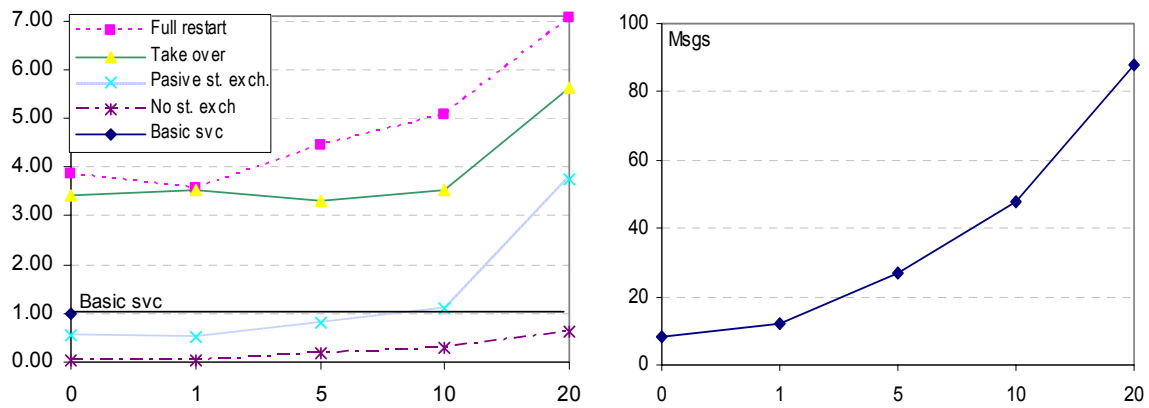


Figure 20: a) MLM recovery time; b) number of messages as a function of the hosted components number

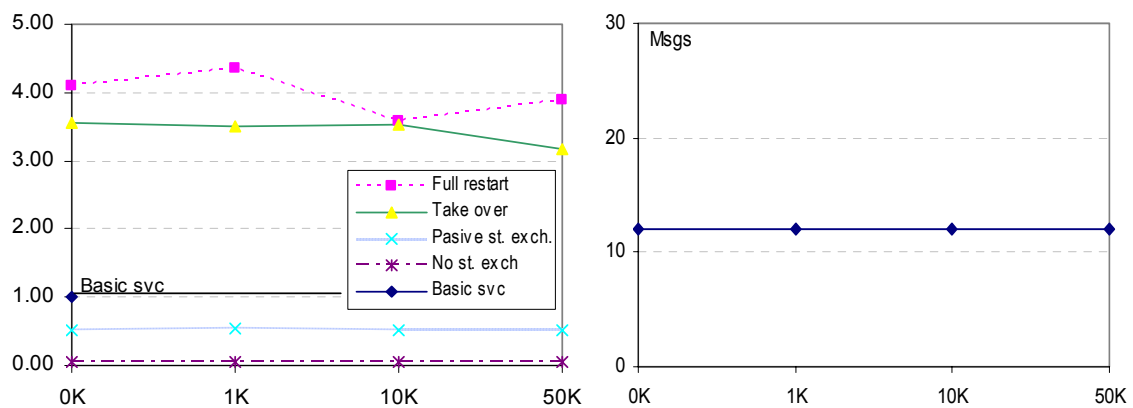


Figure 21: a) MLM recovery time b) number of messages as a function of state size

In order to get a better idea regarding the trade-offs between the HAMSA's benefits and its overhead we compared the behavior of HAMSA framework with that of a simple simulation. This application simulated a monitoring service maintaining its own persistent state backup using the local file system providing very basic reliability semantics with no replication for high availability. We used the recovery time overhead of this application as our basic measurement unit for the performance evaluation. In the above figures, all three recovery time diagrams contain the following four charts:

- MLM full initialization with taking responsibility for the components from the other, already running MLMs, as described in subsection 1a;
- MLM taking over the responsibility for running components with no initialization or state exchange, as described in subsection 2a;
- MLM taking part in the state exchange only, as described in subsection 1b;
- MLM handling a membership change event that does not apply a state exchange, as described in subsection 2b;

7.2.4 Discussion

In the theoretical trade-off analysis of Section 7.1, we assumed that only unicast communication is enabled between the MLMs in HA-MLM groups. However, in our experimental setup we used a single broadcast domain. The implication of this is that the group communication overhead due to failure detection and membership maintenance is reduced by the constant factor of n , where n being the size of the HA-MLM group. Similarly, the application-specific communication overhead of a HAMSA-based application due to the state synchronization is reduced by the same factor. This explains the difference between the theoretical estimations and our experiments results.

In particular, in our first experiment the state replication required a single message to update all the HA-MLM members. However, with no support for multicast, the state replication message would have been sent to every HA-MLM member separately, which in our case would have doubled the number of the state replication messages.

The measurement results show relatively low throughput for a simple messaging interaction. However, first, the presented results include the group communication overhead that does not depend on HAMSA, and is crucial for a highly available replication mechanism. Second, HAMSA was not designed to serve as a general-purpose messaging bus, but rather was focused on providing strict high availability guarantees to the hosted components.

On the other hand, we showed that HAMSA replication mechanism's overhead does not significantly depend on the amount of replications requested by the hosted components. This is an extremely important feature for a replication framework, which, in particular, shows that even with unicast only communication in place we would get similar results.

In the second experiment, we measured the MLM recovery time and the number of transmitted messages that were involved the HA-MLM state exchange protocol. As explained in Section 4.1.4, this protocol requires sending $m \cdot (m + n)$ messages per HA-MLM, where m being the HA-MLM membership size, and n being the number of the state objects

hosted by the specific HA-MLM. In case the multicast communication is available $m + n$ messages should be sufficient.

We can see that the MLM recovery time was affected neither by the HA-MLM membership size, nor by the size of the replicated state objects. The membership size only affected the number of the state advertisement messages sent in the process of state exchange, which, in our case of two HA-MLMs, was minor time-wise, adding just $2m$ messages to the state exchange process (m varied from 3 to 7). However, we could see this overhead in the growing number of messages, which still is reasonable, since this is the minimal number of messages required by any replication protocol.

The tested state objects size did not contribute any additional messages, since a state object could fit into a single message. The only additional overhead in this case was the longer I/O operation of writing the state objects to the file. It seems that for the tested object sizes this issue did not become a bottleneck.

However, the number of hosted components has a significant impact on both the MLM recovery time, and the number of messages sent during the state exchange process. Without getting into the low level details of the HAMSAs implementation, we can see that the overhead, both time and message-wise, grows linearly with the number of the hosted components (notice that in our diagrams the x-axis grows exponentially). This overhead, again, can be considered reasonable, since each component requires a dedicated handling for its state replication, as well as for the execution by its MLM host in a separate thread.

The current HAMSAs implementation's main target was to provide a proof of concept for the feasibility of our approach. We consider this implementation being a prototype, rather than a full-scale mature product. Further code optimization will be required to enhance HAMSAs functionality and to improve its performance.

In spite of these facts we showed that although HAMSAs introduces a significant overhead, this overhead is reasonable and not any higher that one might expect from a generic high-availability and replication framework for network monitoring services. Moreover, we demonstrated that HAMSAs significantly improves the existing model for the primary/backup family of replication protocols, providing an extensive architecture adjusted to the specific requirements of the network monitoring applications and their management.

8 Related work

HAMSA combines different architectural and technology elements: 3-tier architecture, distributed delegation, high availability with the strong semantics.

Achieving scalability of the network management through the hierarchical architecture has been a common approach for a long time. The quest for more efficient and versatile management paradigms has been pursued by many researches over the last few years.

The multi-tier architecture, and in particular, the three-tier architecture approach for monitoring is not new. Starting from the SNMPv2 [11] protocol the multi-tier approach is proposed as part of the SNMP family of standards based on the so-called Mid-Level Managers (MLMs) as well as the Manager-to-Manager (M2M) communication model. These enhancements to the original SNMP framework have a potential to dramatically improve scalability of the SNMP-based monitoring.

Other approaches suggest using mobile agents, active networks, or programmable networks for decentralizing and shortening the control [16], [19]. Usually, these proposals focus on the mechanics of the mobility and extended functionality rather than on the high availability and meta-management issues being in the focus of this work.

Several approaches for integrating the management by delegation approach [24] into SNMP environment have been proposed recently [25]. With the advent of Java, the delegation is easily implemented by exploiting its mobility and security features making Java a preferred language for developing delegated programs.

Novel architectures were proposed in the recent years for highly flexible and adjustable multi-tier management frameworks, such as [26], [27]. However, they assume a weak failure model for their architecture, which cannot satisfy important types of network monitoring services as presented in this work.

Java Management Extension [17] is an emerging Java standard for representing managed objects as Java Beans. JMX Bean is an object that serves as a Java wrapper facade for the actually managed object. JMX Beans can co-locate with the objects they represent at the agent side, or be deployed in a distributed fashion. In the latter case, JMX Beans need distributed object services of the second tier that are currently left unspecified by JMX. HAMSA components can be implemented as JMX Beans.

One of the more mature Java technologies for deploying three-tier Java applications is provided by Enterprise Java Beans (EJB) [21]. EJB defines interfaces for Application Server, and Enterprise Java Bean components that execute in the environment of the application server managing all the transactions, persistency, security, and naming services for the components.

The problems that HAMSA copes with are very similar to those of the state-full EJB clustering. Some of the existing EJB implementations provide fail-over models that allow for replication of the beans' states, and support takeover of the failed beans by other servers in the cluster [21]. Most EJB servers perform state-full fail-over by using either in-memory replication, or persistent storage to a shared database. These solutions are inappropriate for the network monitoring domain, since they rely on the fact that the network remains

connected. To the best of our knowledge, there is no current implementation of EJB, or other application server technology that provide the high availability of the second-tier components execution to the level that allows their comparison with HAMSA.

And finally it is definitely worth mentioning last but not least, the industry standard network management and monitoring tools: OpenView by HP and NetView by IBM. These tools are historically most widely used and are considered as very sophisticated applications in the network management area. However, we believe that, while providing advanced means for flexible distributed management, these tools still do not put enough emphasis on the meta-management issues, and specifically, the high availability of the management and monitoring applications.

9 Future Work

This section suggests some of the possible directions of future HAMSA enhancements.

9.1 Advanced Load Balancing and QoS within HA-MLM

It is possible that due to some load considerations, it will become necessary to move a HAMSA component among the MLMs within its HA-MLM membership even during the components' normal execution, and not just in case of failures. This is in order to prevent the load accumulation on a specific MLM and, thus, the degradation in the *quality of service (QoS)* provided to the management component by its host.

A management component may be interested in requesting a specific QoS level from its HA-MLM. In this case we could consider the following approach: At some point a host MLM concludes that it is only possible to satisfy the requested QoS by moving the management component. Then the MLM would exchange the QoS-related information with other MLMs via the group communication service in order to identify the most appropriate new location for the management component.

In this approach, MLMs do not need to monitor each other continuously, but only when the real need to re-balance the management components arises. Therefore, the overhead of the load balancing is kept reasonably low.

Additional more advanced load-balancing and QoS policies can be considered.

9.2 Automatic HA-MLM Construction

In the current version of HAMSA, a human network administrator specifies the members of a HA-MLM group manually. This makes the management systems based on HAMSA very flexible and provides the administrator with full control over them. On the other hand, as a management system grows and its complexity increases, it becomes time-consuming and failure-prone to manage HA-MLM groups manually. Moreover, human managers may not always choose the optimal set of MLMs to comprise a HA-MLM group with respect to multiple dynamically changing parameters such as network load, resource consumption, load sharing, physical location, etc.

In the future, the work of a HA-MLM group construction could be automated to the possible extent. Ideally, an administrator should be capable of specifying a management component along with its QoS and high-availability requirements, leaving the low-level work of optimal HA-MLM formation to HAMSA, while still having the option to intervene into the construction process manually, if necessary.

10 Conclusion

Efficient monitoring of large and dynamic distributed systems becomes challenging. In spite of numerous novel technologies and approaches described in brief in this work, they are not widely utilized by the management community yet. There are several reasons for this. We believe that one of the more fundamental problems with the distributed hierarchical management in general is the increased complexity of the *meta-management*, *i.e.*, administrating the management system itself.

We present a lightweight monitoring middleware called HAMSA that dynamically allows to enhance monitoring functionality, and to decentralize it in a reliable and efficient manner. This work presents the architectural overview of the middleware, and the possible functional and performance trade-offs involved in its deployment. By leveraging a group communication middleware our architecture increases availability, modularity, and scalability of network monitoring.

Bibliography

- [1] Y. Yemini, *The OSI Network Management Model*. IEEE Communications Magazine pages 20-29, May 1993.
- [2] A. Sahai and C. Morin, *Towards Distributed and Dynamic Network Management*, Proc. NOMS'98, New Orleans, Louisiana, USA, February 1998.
- [3] M. Feridun, W. Kasteleijn and J. Krause, *Distributed Management with Mobile Components*, Proc. IM'99, Boston MA, USA, May 1999.
- [4] A. Bieszczad and B. Pagurek, *Towards Plug-and-Play Networks with Mobile Code*, Proc. ICC'97, Cannes, France, November 1997.
- [5] M. Zapf and K. Herrmann and K. Geihs, *Decentralized SNMP Management with Mobile Agents*, Proc. IM'99, Boston, MA, USA, May 1999.
- [6] ACM, *Communications of the ACM 39(4)*, special issue on Group Communications Systems, April 1996.
- [7] Y. Amir, D. Dolev, S. Kramer and D. Malki, *Transis: A Communication Sub-System for High Availability*, Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing, July 1992.
- [8] Robert Orfali, Dan Harkey, and Jeri Edwards, *Client/Server Survival Guide*, 3rd edition, John Wiley & Sons, 1999
- [9] RADIUS, IETF RFC2138, RFC2139
- [10] M. Dilman and D. Raz, *Efficient Reactive Monitoring*, *IEEE Journal on Selected Areas in Communications (JSAC)*, special issue on recent advances in network management, 20(4):668–677, May 2002.
- [11] William Stallings, *SNMP, SNMPV2, SNMPV3, and RMON 1 and 2*, Addison-Wesley, January 1999.
- [12] K. P. Birman, A. Schiper and P. Stephenson, *Lightweight causal and atomic group multicast*, ACM Transaction on Computer Systems, Vol. 9 (3): 272-314, August 1991.
- [13] BBN Technologies, *Cougaar Architecture Document*, ver. 10.0, February 2003.
- [14] N. Budhiraja, K. Marzullo, F. B. Schneider, S. Toueg, *Primary-backup Protocols: Lower bounds and optimal implementations*, Third IFIP Working Conference on Dependable Computing, Mondello, Italy, pp. 187—198, January 1992
- [15] M. Daniele, B. Wijnen, M. Ellison, and D. Francisco, *Agent extensibility (AgentX) protocol*, RFC 2741, January 2000.
- [16] A. Bieszczad, B. Pagurek, and T. White, *Mobile agents for network management*, IEEE Communications Surveys, 1(1):2–9, 1998.
- [17] Sun Microsystems, *Java management extensions (JMX) instrumentation and agent specification*, v1.1, mar 2002.
- [18] B. Pagurek, Y. Wang, and T. White, *Integration of mobile agents with SNMP: Why and how*, In 2000 IEEE/IFIP Network Operations and Management Symposium, pages 609 – 622, Honolulu, Hawaii, USA, April 2000.
- [19] D. Raz and Y. Shavitt, *Active networks for efficient distributed network management*, IEEE Communications Magazine, 38(3), March 2000.
- [20] M. G. Rubinstein and O. C. M. B. Duarte, *Evaluating tradeoffs of mobile agents in network management*, Networking and Information Systems Journal, 2(2):237–252, 1999. HERMES Science Publications.

- [21] E. Roman, S. Ambler, and T. Jewell, *Mastering Enterprise Java Beans*, Wiley, 2nd edition, 2002.
- [22] E. P. Duarte Jr. and Aldri L. dos Santos, *Semi-Active Replication of SNMP Objects in Agent Groups Applied for Fault Management*, 7th IEEE/IFIP International Symposium on Integrated Network Management IM, Seattle, WA, May 2001
- [23] T. Anker, G. Chockler, D. Dolev, and I. Shnaiderman, *The design of xpanse: A group communication system for wide area*, Technical Report HUJI-CSE-LTR-2000-31, The Hebrew University, July 2000
- [24] Y. Yemini, G. Goldszmidt, and S. Yemini, *Network Management by Delegation*, In The Second International Symposium on Integrated Network Management, pages 95–107, Washington, DC, USA, April 1991.
- [25] D. Levi and J. Shonwalder, *Definitions of Managed Objects for the Delegation of Management Scripts*, May 1999, RFC 2592.
- [26] E. Al-Shaer, *A Dynamic Group Management for Scalable Distributed Event Correlation*, IEEE/IFIP Integrated Management (IM'2001), May 2001.
- [27] R. van Renesse and K. Birman, *Astrolabe: A Robust and Scalable Technology For Distributed System Monitoring, Management, and Data Mining*, Submitted to ACM TOCS, Nov. 2001.
- [28] Sun Microsystems, *Java 2 Platform, Standard Edition Platform Overview*, <http://java.sun.com/j2se>
- [29] T. Berners-Lee, R. Fielding, U.C. Irvine, L. Masinter, *Uniform Resource Identifiers (URI): Generic Syntax*, IETF, RFC 2396
- [30] Leslie Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, Communications of the ACM, 21(7): pp 558-565, 1978

Appendix A: HAMSA Installation Guide

This document will guide you how to install HAMSA in just 10 easy steps.

The HAMSA distribution supplies two basic modules:

- *MLM server*: server-side daemon actually providing the core HAMSA functionality. You should install an instance of MLM on every server that you want to take part in the HAMSA framework.
- *AdminGui*: client-side GUI front-end for easy administration of HAMSA. You should install an instance of AdminGui on any administration station that will deal with HAMSA.

Prior to the HAMSA installation make sure that you have a full JDK 1.3.1 or higher installed on all the machines you want to utilize for the HAMSA framework. If you have already got it you can now proceed with the installation as explained below.

1. Unzip the content of the HAMSA distribution onto your disk.
2. Decide where you want to install HAMSA, and define a new environment variable *HAMSA_HOME* that specifies the desired deployment location path.
3. Locate the installation environment setup configuration script:
 - *env-setup.bat* for Windows OS
 - *env-setup.csh* for UNIX OS
4. Update the selected *env-setup* configuration script as follows:
 - a. Mandatory:
 - Windows only: specify the drive letter of the deployment path, e.g. *D*:
 - b. Optional (for advanced users only):
 - Specify the installation type by changing the *INSTALLATION_TYPE* variable's value to either *MLM* or *Admin*. This will affect the *hamsa.jar*'s main class setting in the manifest file.
 - Specify the deployment path in case you want it to be different from *HAMSA_HOME*.
 - It is recommended to launch the installation process from the very directory, where the installation script is located. Otherwise, specify the distribution location path.
 - In case you prefer your installation to use a temporary directory other than the specified one, you should change the *TMP_VOLUME* (Windows only) and *TMP_DIR* variables' values.
 - In case you prefer using compilation flags different from the default one, you can update the *FLAGS* variable.
5. Update the *local_config.txt* file to be used in the installation. This file should contain a non-empty list of the MLM names (one per line). An MLM name typically consists of the name of the host where the MLM will be executing and the logical name of the

specific MLM. The host name and the logical name are concatenated through the '_' character. Please notice that you should use the host name as it is provided by the system *hostname* call. In addition, the host name should only contain the host part of the fully qualified dotted notation's host name, i.e., without the domain name. Please also notice that the lines starting with the semicolon character are treated as comments lines.

6. Locate the appropriate installation script. You can choose one of the following HAMSAs installation procedures:
 - a. Runtime installation - uses precompiled distribution build (JAR libraries). Launch it by running:
 - *install.bat* for Windows OS
 - *install.csh* for UNIX OS. It uses standard C-shell. You may need to grant execution permissions to this file if missing, in order to be able to run it. Refer to the '*chmod*' command's manual documentation for more information.
 - b. Build installation - allows you for changing the HAMSAs code by rebuilding all the HAMSAs application's JAR libraries. It is currently available for Windows installation only. Launch it by running:
 - *install-build.bat* for Windows OS

Once completed the build installation performing the runtime installation is no longer necessary.

7. Launch the installation script. As mentioned above it is strongly recommended that you do it from the root directory of the distribution, i.e., the very directory where the installation script itself is located.
8. For easier use of HAMSAs it is recommended to add the HAMSAs bin directory to your system path as follows:
 - Windows: *%HAMSAs_HOME%\bin*
 - UNIX: *\$HAMSAs_HOME/bin*
9. Install Transis (see the Transis installation guide for more details).
10. Refer to the HAMSAs Administrator Guide to get HAMSAs up and running.

Enjoy HAMSAs!

Appendix B: HAMSA Administration Guide

This HAMSA distribution supplies two basic modules:

- *MLM server*: server-side daemon actually providing the core HAMSA functionality. You should have a running MLM daemon instance on every server that you want to take part in the HAMSA framework.
- *AdminGui*: client-side GUI tool for easy administration of HAMSA. One can decide to develop her own administration client using the HAMSA's extensive APIs.

MLM server

In order to launch an MLM server daemon you should:

1. Set up the MLM configuration by updating the *config.txt* file located in the *mlm* sub-directory of the *HAMSA_HOME* directory.

Please notice that all the MLMs' configuration files should contain consistent MLM daemons section listing the same MLM names for all the participating hosts.

2. Set up the Transis daemons configuration by updating the Transis configuration file at all the hosts comprising your HAMSA framework. The Transis configuration file is typically named '*config*' and is located in the Transis '*bin*' directory.
3. If it is not the first time you are launching the MLM and you are using the "*-restore*" option, you may want to reset the backup MLM state by deleting its directory located under the *mlm* sub-directory of the *HAMSA_HOME* directory. The MLM state directory name is equal to the full MLM daemon name including both the host name part and the logical name part, e.g., *HOST_MLM0*.
4. Launch Transis daemon or ensure that a daemon is already running (one per host). See the Transis user guide for more details.
5. Launch the RMI Registry daemon using the provided script:

- Windows:
`%HAMSA_HOME%\bin\RMI.bat`
- UNIX:
`$HAMSA_HOME/bin/RMI.csh`

6. Launch the MLM daemons on all the appropriate hosts as follows:

- Windows:
`%HAMSA_HOME%\bin\MLM.bat -name <MLM_logical_name> -restore`
- UNIX:
`$HAMSA_HOME/bin/MLM.csh -name <MLM_logical_name> -restore`

Please notice that the MLM logical names should only contain the logical part of the MLM name specified in the configuration files, i.e., the host name part should be

omitted. In case you omit the "-name" option the default "MLM0" logical name will be used.

Administration tool

1. Set up the AdminGui configuration by updating the config.txt file located in the admin sub-directory of the HAMSA_HOME directory. Please notice that the AdminGui client's configuration file should be set up appropriately containing the MLM daemons configuration similar to that of the MLMs' configuration files.
2. Launch an AdminGUI client as follows:
 - Windows:
`%HAMSA_HOME%\bin\AdminGui.bat`
 - UNIX:
`$HAMSA_HOME/bin/AdminGui.csh`

Notes

- Please notice that Windows OS users can also benefit the shortcuts supplied in the HAMSA's bin sub-directory.
- Please refer to the HAMSA User Guide for the details of the HAMSA core features.

Enjoy HAMSA!