Reachability Snapshots in the Presence of Failures: An Exercise in Protocol-Service Composition

Sebastian Gutierrez-Nolasco and Nalini Venkatasubramanian
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{seguti,nalini}@ics.uci.edu

Abstract

When two services are composed in order to obtain their combined benefits, their composition is constrained in order to prevent functional interference between them and with other services of the system. However, middleware services make assumptions about the underlying communication environment. If these assumptions are not satisfied by the communication subsystem, correctness violations, such as inconsistent states and incorrect execution semantics may occur.

This paper explores one specific instance of protocol-service composition that guarantees the safe execution of a snapshot service in the presence of a fault-tolerance protocol.

1 Introduction

In this paper, we address the composition of resource management services with communication protocols. Such composition must address the following issues: (1)Preserve individual service semantics and functional non-interference of middleware services and (2)Preserve (individual and composite) protocol guarantees. In order to achieve this, we may need to constrain the composition of middleware services with communication protocols to ensure safe service-protocol composition. Safe service-protocol composition is required to protect the system from reaching inconsistent states that can lead to deadlocks, livelocks and incorrect execution semantics.

Let us consider the concurrent execution of a reachability snapshot service in an environment prone to node failure. The reachability snapshot service allows us to gather relevant and accurate information about the reachability status of objects in a distributed environment. This helps in making decisions like checkpointing or (distributed) garbage collection. Since nodes may crash and recover, the system decides (at some point of time) to introduce a failure detection and recovery protocol in order to detect node crashes as quickly and accurately as possible and recover from them. In general, failure detection protocols in distributed systems are based on a heartbeat mechanism, where nodes exchange information about their current status. Techniques have been developed to design failure detectors in distributed systems that can guarantee completeness (the failure of a node is eventually detected) and accuracy (no non-faulty node is considered faulty).

Concurrent execution of the reachability snapshot and the failure detection and recovery protocol may cause problems. For instance, if a node crashes while a snapshot is in progress, restarting the node will not preserve the individual service semantics of the snapshot service. This may in turn, block the snapshot

in progress and introduce some inconsistencies in the reachability view.

In this paper we develop an algorithm to enforce functional non-interference between the reachability snapshot service (a middleware service) and the failure detection and recovery protocol (a communication protocol) and analyze the overheads involved. We use a two level meta architecture model to specify the reachability snapshot service and the failure detection protocol, reason about their properties and interaction between them.

The Two Level Meta Architectural Model

Since the actor model of computation [5] incorporates the notion of encapsulation and interaction only via message passing, it offers a clear, flexible and simple semantic approach to describe distributed systems based on incoming communications [7]: The system is modeled as a group of self contained, and independent autonomous objects called actors, which communicate via asynchronous (buffered) message passing. An actor has a mail address, a behavior and it is inactive until someone sends it a message. On receiving a communication, an actor processes the message in a manner determined by its current behavior.An actor can send a message to another actor only if it knows the mail address of the recipient. Furthermore, mail addresses may be communicated in messages. Thus, the configuration of the communication is dynamic and the activation order (one message activates another if the latter is sent during the processing of the former) determines communication patterns.

The two level meta architecture model (TLAM) refines the actor model to specify, compose and reason about resource management services in open distributed systems. In the TLAM, a system is composed of two kinds of actors, base actors and meta actors, distributed over a network of processing nodes. Base actors carry out application level computation, while meta actors are part of the run-time system, which manages system resources and controls the run-time behavior of the base level. Meta actors communicate with each other via message passing as do base actors, and they may also examine and modify the state of the base actors located on the same node. Base level actors and messages have associated run-time annotations that can be set and read by meta actors, but they are invisible to base level computation. Actions, which result in a change of base level state, are called events and meta actors may react to them if they occur on their node.

A TLAM configuration has a set of base and meta level actors and a set of undelivered messages. The actors are distributed over the TLAM nodes. Each actor has a unique name (actor identifier) and the configuration associates a current state to each actor name. The undelivered messages are distributed over the network (some are traveling along communication links and others are held in node buffers). The semantics of a TLAM is given by a labeled transition relation on configurations. These transitions are described by reaction rules specific to a particular TLAM that determine how individual actors react to messages received, and in the case of meta actors how they react to events.

2 Reachability Snapshot Service

Reachability is a fundamental concept in actor systems because it characterizes the potential for communication of one actor with another. It forms the basis for resource management activities such as distributed garbage collection and checkpointing. In particular, a reachability snapshot service records the set of reachable (base level) actors of a particular configuration.

We use the following terminology to specify the reachability service [11]. We consider reachability from a *root set*, a pre-defined set of actors which are to be considered independently reachable. We define the *acquaintances* of an actor in a configuration as the actor identifiers contained in the actor state plus those contained in undelivered messages to that actor. Similarly, an actor is considered *busy* or *enabled* in a configuration if the actor is processing a message or if there is an undelivered message

to that actor. The *inverse acquaintance* of an actor in a configuration are the actors that have that actor as an acquaintance in the configuration[1]. An actor is considered *inactive* in a configuration if it is not *busy* and *permanently inactive* if it is inactive and it is not connected to a busy actor via some chain of inverse acquaintances. Using these informal notions, the set of reachable actors in a configuration is the least set (of base level) actors in that configuration such that: (1)Every root actor (*ra*) is reachable, (2)Every forward acquaintance (*fa*) of a reachable actor is reachable and (3)If an actor is reachable, then every inverse acquaintance (*ia*) of that actor which is not permanently inactive is reachable.

However, in order to represent the global state of a distributed actor system, we need a mechanism for recording the state of all nodes including the portion of the node being communicated in the network channels [2]. This is accomplished by forming a logical two dimensional grid with the nodes of the system and using two wave protocols for message propagation that (a)visit all nodes exactly once, capturing the node resident information and (b)traverse all links in the system exactly once forcing messages on channels to reach nodes, where their state can be recorded.

In order to achieve this, each node has a designated set of forward and backward neighbor links as well as a set of broadcast predecessors and successors, forming a partial order between nodes. We introduce two kind of messages broadcast messages and bulldoze messages. These messages are used to synchronize and clear the nerwork respectively. Hence, a node can issue a broadcast message to its broadcast successors only after it has received the message from all of its broadcast predecessors. There are two kinds of bulldoze messages (forward bulldoze and backward bulldoze messages). Forward bulldoze messages are propagated through the forward neighbor links; while bulldoze messages are propagated using the backward neighbor links. The starting and termination points of both waves, the propagation paths and constraints on message propagation are defined in [11]. This model assumes that the network consists of channels linking pairs of nodes. Each channel consists of a pair of directed links (one in each direction) with infinite message buffers and it requires that message order is preserved across a single link[2]. Hence, the system as a whole may be:

- Stable, where every node is unaware of any snapshot in progress and maintains the last recorded snapshot information.
- Initializing a new recording, where some nodes are begining to record a local snapshot.
- Finalizing a recording, where the recording is completed.

Locally, a given node may be in one of three phases:

- **Unaware (U)**, where the node is unaware of any snapshot in progress.
- **Begining local recording (B)**, where the node begins to record a local snapshot.
- **Finalizing local recording (F)**, where the node finalizes the local recording of a snapshot.

A global-Snap meta actor coordinates the global snapshot process and local-Snap meta actors in each node of the system coordinate and gather the local snapshot information on each node. In the initial configuration, the system is stable and every node in the system is in phase **U** and every local-Snap has a list of root actors running on its node (*ralst*). When a new snapshot request is accepted by the global-Snap meta actor:

1. The global-Snap will generate a broadcast message (*BC(init)*) to notify every local-Snap that a new snapshot is in progress.
2. Upon receipt of the broadcast message, every local-Snap moves to phase **B**, where it prepares itself to record the information as soon as it becomes locally available (delivery of messages to local base actors).

[1] Since a non-reachable actor may become reachable by communicating (at some point of time) its address to a reachable actor, inverse acquaintances are crucial to determine actor reachability

[2] Since communication between two different nodes may need to traverse several links, this assumtion does not imply in-order arrival of messages at their destination

3. The global-Snap proceeds to send the first *forward bulldoze* (Fb) wave (*see figure 1*) to force current messages on channels (*old messages*) to reach nodes, where their state can be recorded.

4. Upon receipt of the Fb message, the local-Snap marks every actor with a empty mail queue that is not currently processing a message as inactive and tags every outgoing message as a *new message*, indicating that its information has been recorded at the sender side.

5. Once the local-Snap of the *last node* receives and processes the Fb message, it sends an *end()* message to the global-Snap. Then, the global-Snap proceeds to send the first *backward bulldoze* (Bb) wave (*see figure 2*), which will clear *old messages* still floating in the network.

6. Upon receipt of the Bb message, the local-Snap can finish the construction of the forward aquaintance list of its actors (*sfa*) and begin the construction of its inverse acquaintance list (*sia*) by sending *I-know-you()* messages to every actor that appears in any forward acquaintance list.

7. Every time a local-Snap receives an *I-know-you()* message, the address of the sender is added to the target actor's inverse acquaintance list (*sia*).

8. When the local-Snap of the *first node* receives the Bb message, it sends an *end()* message to the global-Snap. Then, the global-Snap sends a second set of *forward bulldoze* and *backward bulldoze* waves in order to clear the network of *I-know-you()* messages.

9. As soon as the second *backward bulldoze* wave finishes, the global-Snap broadcasts a message to every local-Snap, indicating that the network is clean and the recording is complete.

10. Upon receipt of this message, the local-Snap can determine if an *inactive* actor is permanently inactive and generates a transition to phase **F**, where the new snapshot information is stored and all the unnecessary information is cleared.

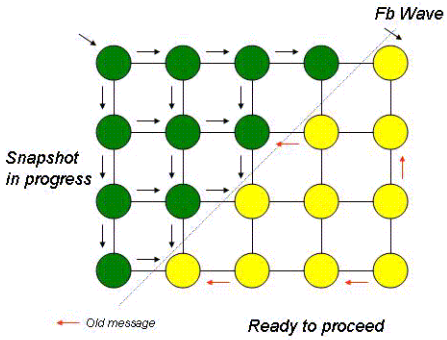

Figure 1: The forward bulldoze wave partitions the network grid in two regions: where the local snapshot is in progress and where the nodes are waiting the Fb message to begin the local recording.

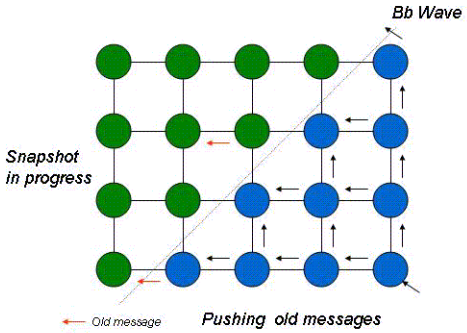

Figure 2: Since old messages may be traveling in the inverse direction of the Fb wave, a backward bulldoze wave is needed to push the remaining old messages out of the channels.

Note that the phase change is performed asynchronously at every node. Once a node transitions from phase**U** to phase **B**,it does not know whether the entire system is aware of the snapshot request. At the end of the initialization all nodes are in phase **B** and each node is notified of that fact. This notification moves a node from phase **B** to phase **F** during which recording is completed. Since every node needs to return to phase **U** when a local snapshot is complete, the global finalization has two parts: (1)Notification of finalization during which nodes are in phase **B** or **F** and (2)Completion of finalization during which nodes are in phase **F** or **U**.

3 Failure Detection Protocol

Reliability is an important issue in distributed systems, since it is difficult to guarantee that the applications work properly in presence of node failures. Traditionally, failure detection protocols have been used to provide *hints* about possible failures in the system. However, they are often entwined with the distributed algorithms using them, making it difficult to reason about the algorithm without taking into account its implementation or modifying the failure detection protocol without modifying the dirtibuted algorithm. In recent years, failure detection protocols have been designed as modular components that can be integrated and interchanged without impacting the algorithm using them.

In general, failure detection protocols in distributed systems are based on a heartbeat mechanism, where nodes exchange information about their current status. The main advantage of this scheme is that it guarantees completeness (the failure of a node is eventually detected). However, node failure information can be inconsistent if it is based solely on *one* timeout or timeouts from one particular node, since a transient overload could cause some node to be considered unavailable while it continues to execute. As a result, several techniques have been developed to design failure detectors that can achieve some degree of accuracy (no non-faulty node is considered faulty) and efficiency (the node failure detection time is bounded), depending on the heartbeat dissemination mechanism used.

Failure detectors that guarantee completeness deterministically while achieving efficiency and accuracy only probabilistically ([15] [13] [1]), have been able to circumvent the impossibility of achieving both completeness and accuracy in asynchronous networks prone to failures [14].

We adopt the randomized failure detection algorithm presented in [8] to implement our failure detection protocol due to its high accuracy and speed of failure detection (on average). The algorithm detetcs a node failure within an expected time, τ, instead of an exact time and achieves an accuracy probability based on this expected time. Although the algorithm does not need clocks to be synchronized, it certainly requires each node to have a steady clock rate that allows accurate measurement of a protocol period τ (in time units). Hence, at each non-faulty node N_j the following steps are executed every τ units:

1. Select a random node N_i, send a ping message to it and wait for the worst case message roundtrip time for an acknowledgement message from N_i.
2. If timeout expired and the acknowledgement from N_i was not received, select a group of nodes and request them to send a ping to N_i on behalf of N_j and reset the timeout.
3. If timeout expired for a second time and the acknowledgement from N_i has not been received, declare node N_i as crashed.

Note that at anytime, a node N_k may receive 1)a request to ping N_i on behalf of N_j or 2)an acknowledgement of N_i that needs to be forwarded to N_j

4 Snapshot and Failure Detection Composability

Usually, when two services are composed in order to obtain their combined benefits, their composition is constrained in order to prevent functional interference between them and with other services in the system. However, enforcing functional non-interference between services and protocols is not sufficient to obtain safe composition of protocols and services due to the fact that middleware (resource management) services make assumptions about the underlying communication environment. If these assumptions are not satisfied by the communication subsystem, correctness violations, such as inconsistent states and incorrect execution semantics may occur.

Let us now consider the concurrent execution of the reachability snapshot service and the failure detection protocol described in the previous section. Since distributed systems are loosely coupled, in terms of their relative speed and local activities, a global clock is usually not possible. However, the dynamicity of the actor model requires some notion of synchronization in order to obtain a consistent distributed

reachability snapshot. This notion of synchronization is achieved through the use of wave protocols for message propagation, which simulate synchronization barriers. The failure detection protocol ensures information consistency of the crashed node, but it makes no guarantees about its state information when it recovers and reconnects to the logical network created by the snapshot service.

A node failure can affect the snapshot service in progress even if the crashed node is restarted. *i.e.* if a node crashes while a wave protocol for message propagation is in progress, merely restarting the node can not determine if the wave protocol message was delivered to the node before crashing, after crashing or it is floating through the channel and will be delivered soon. Thus, the restarted node must know (1)the snapshot phase in it should be (**U**, **B** or **F**),(2)if there is a wave protocol in progress and the specific kind of wave protocol (*forward bulldoze* or *backward bulldoze*), and (3)how to catch up with the rest of the (non-faulty) nodes in the next synhcronization barrier.

In our approach, we add fault tolerance to the reachability snapshot service by enhancing the failure detection protocol and refining the reachability snapshot algorithm to use wave protocols as synchonization barriers, where recovered nodes can reach a consistent state and integrate with non-faulty nodes.

Enhanced Failure Detection Protocol

Our model assumes a crash-failure model, where no byzantine failures occur, node failures are independent and messages are delivered uncorrupted in a reliable medium. System nodes have assigned a logical unique identifier (*nodeID*), used to form the two dimensional grid explained in section 2, and a monotonic incarnation sequence number (*isn*) used to identify how many times the node has crashed during a given time span[3]. That is, every time a node crashes, it recovers subsequently and increments its *isn*. Note that between synchonization barriers, the *isn* is reset to zero.

Since in the crash-failure model a node failure is indistinguishable from a slow response (due to an asynchronous network prone to failures) and we are interested in node recovery after a crash, we use a failure detector protocol that *suspects* the crash of a node if the node's *isn* is greater than zero (the default *isn* value). Although this approach does not bound node failure detection time, it is possible to detect node crashes and recoveries within a known maximum time by integrating a timed perfect failure detector based on hardwdare watchdogs [3].

The failure detection protocol consists of a set of node failure detection meta actors (*nfd*): Every node in the system has a *nfd* running and every *nfd* can query another *nfd* every τ time units to see if it is alive and has not crashed (*see figure 3*):

- Every τ (local) time units in a node N_i, the nfd_i module running on N_i
  1. Requests the isn of its logical neighbors[4] by issuing a *isnReq(N_i)* message to each one of them and waits for a response for each message sent. The timeout of the wait is the worst-case message roundtrip time.
  2. If a timeout from a node N_k expired and its response has not been received, N_i selects from its logical neighbor list the nodes within a 1 link distance from N_k and request them to contact N_k on behalf of N_i by sending them the message *fwdIsnReq(N_i,N_k)*
  3. If no response message is received from N_k by the end of 2τ after the second message, N_k is declared failed.
- Any time node N_j responds to incoming messages as follows:
  1. Upon receiving an *isnReq(N_i)* message,the node replies with a *isn(getIsnValue(N_j),N_j)* message to N_i.

[3]We assume the isn is a persistent value

[4]We define as logical neighbors all the nodes that are logically one hop distant from the node. Since the nodes form a grid, neigbors vary from 2 to 4 depending on the node position

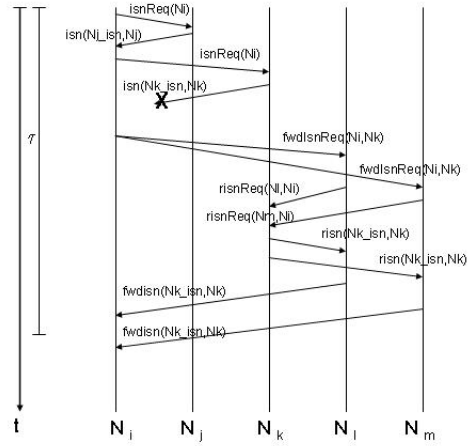

Figure 3: Sequence diagram of the enhanced failure detection protocol:

2. After a *fwdIsnReq(N_i,N_k)* message is received, send a *rIsnReq(N_j,N_i)* message to N_k
3. After a *rIsn(getIsnValue(N_k),N_k)* message is received, send a *fwdIsn(getIsnValue(N_k),N_k)* message to N_i

Since nodes are equipped with a stable storage medium, a crashed node loses only volatile memory and non-volatile memory is recovered. However, in order to guarantee information consistency when a node fails and recovers, the protocol initially assumes: (1)the set of actors on the node does not change between node failure and recovery, (2)the set of messages targeted to actors belonging to the crashed node, will be floating around until the node recovers and then will remain undelivered in the network until the node recovers and then will be delivered to the node and (3)the communication between nodes is reliable and FIFO.

Refined Reachability Snapshot Service

Since the logical 2D grid establishes a node partial order and the wave protocols simulate communication synchronization barriers, we propose to enhance the local-Snap meta actor with: (1)a list of its logical parents (*lplst*), (2)a list of its logical children (*lclst*), (3)a wave protocol counter (*wpc*), (4)an I-know-you flag (*iky*) and (5)a local incarnation sequence number (*lisn*).

In the initial configuration of the system, the parents and children of each node are determined according to the node partial order, the *wpc*, the *iky* and the *lisn* are initialized to zero and every time a node receives a broadcast or bulldoze message, its *wpc* is incremented and its *lisn* is reset to zero. That is, we keep track of node failures only between synchronization barriers. Note that the *iky* flag allow us to determine when the information for the snapshot is being established (when the flag is zero) and when it is being exchanged (when the flag is set).

In case of a node failure, a recovered node increments its *lisn* and proceeds to determine its phase and what kind of wave protocol is in progress as follows[5]:

1. Requests the *lisn, phase and wpc* of its parents and children
2. if the *lisn* of one of the nodes is set, its information is discarded since the node is in the recovering process. This may happen if neighboring nodes also fail within a short time of the failure of N_i
3. Since nodes may only be in different consecutive snapshot phases, a recovered node can determine its snapshot phase by comparing the information of its parents and children as follows:
   (a) If the phases of children and parents are the same, the system is in a stable phase and the recovered node's phase is the same as the parents' phase.
   (b) If the parents' phase is **B** and at least one of the children's phase is *U*, the system is being initialized for a new snapshot and the recovered node's phase is **B**.
   (c) If the parents' and children's phases are either **F** or *B*, the system is being notified of the snapshot finalization and the recovered node's phase is **F**.
   (d) If the parents' and children's phases are either **F** or **U**, the snapshot is complete and the recovered node's phase is **U**.

[5]Here we assume that only one node can fail at a time

4. If the recovered node is in phase B, it can determine how to catch up with the rest of non-faulty nodes by comparing the following information:

   (a) If children and parents *wpc*s are the same, the recovered node reached the synchonization barrier before it crashed and no further task is needed[6]

   (b) If the parents' *wpc* is greater than one of the children's *wpc*, we can assume that a forward bulldoze wave is going on and

      i. If the *iky* is not set, the recovered node proceeds as it just received the first Fb message. That is, it sends Fb messages to its children and marks inactive actors.

      ii. If the *iky* is set, the recovered node proceeds as it just received the second Fb message. That is, it sends Fb messages to its children and continues to build its inverse acquaintance list.

   (c) If the children's *wpc* is greater than one of the parents' *wpc*

      i. If the *iky* is not set, the recovered node proceeds as it just received the first Bb message. That is, it sends Bb messages to its parents, sets *iky* and begins to send *I-know-you()* messages.

      ii. If the *iky* is set, the recovered node proceeds as it just received the second Bb message. That is, it sends Bb messages to its parents and finishes building its inverse acquaintance list.

Although we do not deal with the mechanism for detecting termination of the various phases of the snapshot in this paper. These can be assumed to be strictly meta level interaction requiring some additional assumptions about the network topology and routing mechanisms. Details for one approach can be found in [12].

[6] Although a wave may be in progress, it has not hit the parents

Since a recovered node may change its phase, without receiving a message using the information obtained from its parents and children, it is possible that a phase change message in on the way. Thus, a recovered node must be able to handle duplicated messages. In order to achieve this, every message is tagged with a monotonic *wpc* number and the node keeps track of the *wpc* number of the last processed message.

5 Concluding Remarks

The reachability snapshot service has been used as a basis for distributed garbage collection for active objects [12] [4] [9]. However, node crashes, message losses and duplications complicate the consistency of the snapshot service and demand a fault tolerance module that could be combined with the snapshot service to automatically create a fault tolerant snapshot service. Although this approach may be feasible in certain cases [6] [10], the semantic differences between how the global snapshot is obtained (*i.e.* the snapshot algorithm) affect what the fault tolerance module must do. Furthermore, the cost of adding fault tolerance varies according to the type and number of failures to be tolerated.

In our approach, additional synchronization is needed to ensure safe composability of the failure detection protocol and the snapshot service. Furthermore, minor modifications in both service and protocol are required. However, the addition of synchronization cost is compensated by obtaining a more precise state, which implies less recovery time.

References

[1] C. Fetzer and F. Cristian. Fail Awareness in Timed Asynchronous Systems. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, 1996.

[2] Chandy K.M. and Lamport L. Distributed Snapshots: Determining Global States of a Distributed System. In *ACM Transactions on Computer Systems*, 1985.

[3] Christof Fetzer. Enforcing Perfect Failure Detection. In *Proceedings of the International Conference on Distributed Computer Systems*, 2001.

[4] D. Washabaugh and D. Kafura. Distributed Garbage Collection of Active Objects. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991.

[5] Gul Agha. *Actors: A model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[6] Gul Agha and Daniel Sturman. A Methodology for adapting to patterns of faults. In *Foundations of Dependable Computing: Models and Frameworks for Dependable Systems*, 1994.

[7] Gul Agha and Ian A Mason and Scott F Smith and Carolyn Talcott. A Foundation for Actor Computation. *Functional Programming*, 1993.

[8] Indranil Gupta, Tushar D. Chandra and German S. Goldszmidt. On Scalable and Efficient Distributed Failure Detectors. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing*, 2001.

[9] Isabelle Puaut. A Distributed Garbage Collector for Active Objects. In *Conference on Parallel Architectures and Languages Europe (PARLE'94)*, 1994.

[10] Jean-Charles Fabre and Tanguy Perennou. A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach. In *IEEE Transactions on Computers, (47):78-95*, 1998.

[11] Nalini Venkatasubramanian and Carolyn Talcott. Integration of Resource Management Activities in Distributed Systems. Technical Report Computer Science Department, Stanford University, 1995.

[12] Nalini Venkatasubramanian, Gul Agha and Carolyn Talcott. Scalable Distributed Garbage Collection for Systems of Active Objects. In *International Workshop on Memory Management*, 1992.

[13] Robbert van Renesse, Y. Minsky and Mark Hayden. A Gossip-Style Failure Detection Service. In *Proceedings of the International Conference and Distributed Systems Platforms and Open Distributed Processing*, 1998.

[14] Tushar D. Chandra and Sam Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, 1991.

[15] Wei Chen, Sam Toueg and Marcos K. Aguilera. On the Quality of Service of Failure Detectors. In *Proceedings of the 30th International Conference on Dependable Systems and Networks*, 2000.