Evaluation of Dependable Layered Systems with Fault Management Architecture

Olivia Das, C. Murray Woodside
Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada
email: odas@sce.carleton.ca, cmw@sce.carleton.ca

ABSTRACT

The need for a separate fault-management system, that is able to carry out both failure detection and reconfiguration, is becoming imperative due to the increasing complexity of fault-tolerant distributed applications. Such practice would eliminate the intricacies of the failure detection mechanisms from the application and would avoid repeating them in every program. The dependability of such an application depends on the interconnection of components in the fault-management system, management subsystem failures, delays incurred due to system reconfiguration and failure information propagation in the management architecture, as well as on the structure of the application itself. This position paper describes avenues for evaluating the dependability of a multi-layered service system that uses a separate fault-management architecture.

1. INTRODUCTION

Distributed software systems are usually structured in layers with some kind of user-interface tasks as the topmost layer, making requests to various layers of servers. Client server systems and Open distributed processing systems such as DCE, ANSA and CORBA are structured this way. [2, 1, 12] introduced an approach to express the layered failure and repair dependencies in these systems. However, the work done there is limited by the assumption of instantaneous perfect detection and reconfiguration, and independent failures and repairs.

This position paper describes avenues to incorporate the effect of fault management architecture in the dependability evaluation of layered systems. The fault management architecture influences the dependability in the following ways:

- management component failures and the interconnections among the management components affects the successful system recovery.
- delays for system reconfiguration and detection propagation in the management architecture increases the system downtime.

Our earlier work in [3] considered the delays for detection and reconfiguration by a separate detection architecture for layered systems. However, it was restricted to a particular detection architecture that would support full coverage of the failures by the system. If arbitrary connections among fault management components are considered, then it is possible that due to the loss of connectivity in the management architecture, the system may not be able to detect a failure and therefore would fail even if adequate redundancy exists. This issue has been addressed in this work that extends the work in [3] by taking into account arbitrary fault management architectures.

Other work analyzes the effect of software architecture (and not the management architecture) on reliability and is given by Trivedi *et. al*. [4, 5].

As in [3], this work considers only crash-stop failures, in which an entity becomes inactive after failure, and not to the other more complex failure modes such as Byzantine failure [9].

2. LAYERED MODELS CAPTURING FAILURE OCCURRENCE AND REPAIR BEHAVIOR

Figure 1 shows an example of a layered model using a notation proposed in [1, 2] with two groups of users (50 UserA users and 100 UserB users, which may be people at terminals or at PC workstations) accessing applications

which in turn access back-end servers. The rectangles in this figure represent tasks (i.e. operating system processes) such as AppA or Server1 with entries, which are service handlers embedded in the task. For instance, eA-1, eB-1 are entries of task Server1. An arrow represents a request-reply interaction, such as an RPC. Processors are represented by ovals. The numbering #1, #2 on the request arcs indicate primary/backup choice for a service. Server1 is the primary server while Server2 is the backup, implying that if Server1 fails, both "serviceA" and "serviceB" would use Server2 until Server1 is working again. Failure and repair rates are provided for each component (either a task or a processor).

The special property of multi-layered client-server systems is that a failure of a task or a processor in one layer can cause many tasks, requiring its services, to fail, unless they have a backup. The model in Figure 1 captures such cascaded service operational dependencies.

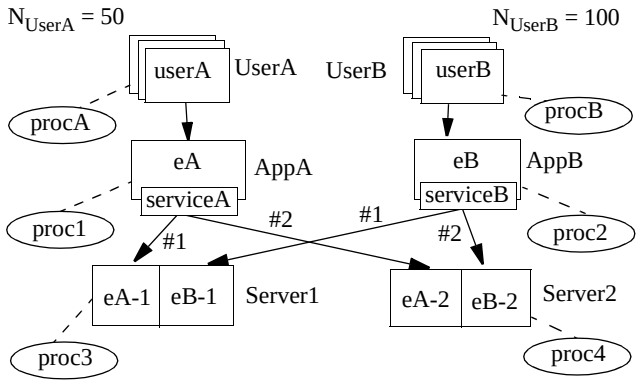

Figure 1. A layered model of a client-server system with two groups of users. Server2 is the backup of Server1.

In order to capture the effect of fault management architectures, the first step would be to describe the architecture in some relevant way. The next section introduces an architectural model to describe various fault management architectures.

3. FAULT MANAGEMENT ARCHITECTURE

The generic management components and their relationships can be depicted as in Figure 2, following [7]. Applications have embedded modules (Subagents) which may be configured to send heartbeat messages in response to timer interrupts (indicating they are alive) to a local Agent, or to a manager directly. A node may have an Agent task which monitors the operating system health status and all the processes in the node, and there may be one or more Manager tasks which collect status information from agents, make decisions, and issue notifications to reconfigure. Reconfiguration can be handled by a subagent (to cause a task or an ORB to retarget its requests) or an agent (to restart a task, or reboot a node altogether).

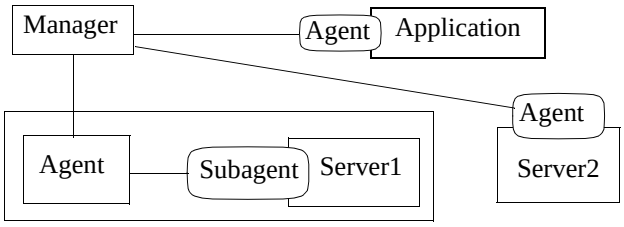

Figure 2. Management components and relationships

The agents and managers are described in this paper as if they are free-standing processes, even though in practice some of these components may be combined with other components in a dependability ORB [8], or an application management system [11].

Failures of system entities are detected by mechanisms such as heartbeats, timeouts on periodic polls, and timeouts on requests between application tasks. Heartbeat messages from an application task can be generated by a special heartbeat interrupt service routine which sends a message to a local agent or to a manager, every time an interrupt occurs, as long as the task has not crashed. Heartbeat messages for an entire node can be generated by an agent configured similarly, to show that the node is functioning; the agent could query the operating system health status before sending its message. Heartbeat information once collected can be propagated among the agents and managers to act as a basis for decisions, made by reconfiguration modules.

An entity that cannot initiate heartbeat messages may be able to respond to messages from an agent or manager; we can think of these as status polls. The responses give the same information as heartbeat messages. Polls to a node could be implemented as pings, for instance.

3.1. Reconfiguration

In this paper, we considered primary-backup replication for achieving fault-tolerance, i.e. the requests are routed to the backup server when the primary server fails, for masking the failure. This alternative targeting of requests is indicated in Figure 1 by showing an abstraction called "serviceA" and "serviceB" for the data access service required by the applications. This service has alternative

request arrows attached to it, with labels "#n" showing the priority of the target. A request goes to the highest-priority available server, which is determined by a *reconfiguration decision*. In this work, the reconfiguration decision will be made by the management system, and will be conditioned by its knowledge of status of system components. It can respond not only to processor failures but also to software failures (task crashes and operating system crashes). Network components can be included in the model as well. A reconfiguration strategy different from the alternative targeting of requests to the highest-priority available server can also be analyzed. For instance, a strategy which involves distributing the workload equally among the available servers can also be considered.

3.2. Management Architecture

The architecture model described here will be called MAMA, ***M***odel for ***A***vailability ***M***anagement ***A***rchitectures. The model has four types of components: application tasks (which may include subagent modules), agent tasks, manager tasks, and the processors they all run on (network failures are for the time being ignored). There are three types of connectors: *alive-watch*, *status-watch* and *notify*. These connectors are typed according to the information they convey, in a way which supports the analysis of knowledge of the system status at different points in the management system.

Components have *ports* which are attached to connectors in certain *roles*. The roles are defined as part of the connector type. The connector types and the roles they support are:

- *Alive-watch* connectors, with roles *monitor* and *monitored*. They only convey data to detect crash failure of the component in the *monitored* role, to the component in the *monitor* role. A typical example is a connector to a single heartbeat source.
- *Status-watch* connectors, also with roles *monitor* and *monitored*. They may convey the same data about the *monitored* component, but also propagate data about the status of other components to the component in the *monitor* role. A typical example is a connector to a node agent, conveying full information on the node status, including its own status.
- *Notify* connectors, with roles *subscriber* and *notifier*. The component in the *notifier* role propagates status data that it has received to a component in a *subscriber* role, however it does not include data on its own status.

Manager and Agent tasks can be connected in any role; an Application task can be connected in the roles *monitored*, or *subscriber*. A Processor is a composite component that contains a cluster of tasks that execute there. If the processor fails, all its enclosed tasks fail. The Processor can only be connected in the *monitored* role to an *alive-watch* connector (which might convey a ping, for example).

Upon occurrence of a failure or repair of a task or a processor, the occurrence is first captured via *alive-watch* or *status-watch* connections and the information propagates through *status-watch* and *notify* connections, to managers which initiate system reconfiguration. Reconfiguration commands are sent by *notify* connections. Cycles may occur in the architecture; we assume that the information flow is managed so as to not cycle. In this work, we note that if a task watches a remote task, then it also has to watch the processor executing the remote task, in order to distinguish between the processor failure and the task failure.

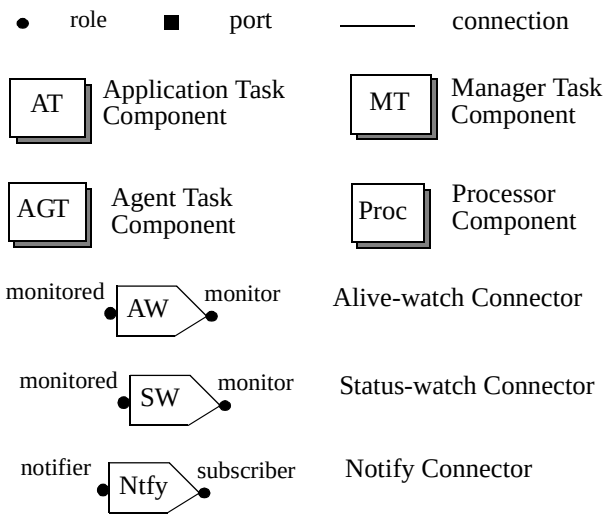

Figure 3. MAMA notations. The graphical notation of components, ports, connectors and roles are taken from [6].

Figure 3 shows a graphical notation for various types of components, ports, connectors and roles based on the customized UML notation for conceptual architecture as defined in [6]. The component types and connector types will be shown as classes in this work. In order to avoid cluttering in the MAMA diagrams, the role names such as *monitor*, *monitored*, *notifier* and *subscriber* have been omitted from them.

Figure 4 shows a centralized management architecture, in MAMA notation, for the system of Figure 1. Manager1 is introduced here as the central manager task that collects status information from the agents ag1-ag4 running on the

processors proc1-proc4. The application tasks AppA and AppB are also subscribers for the notifications from Manager1, which control retargeting of requests to the Servers.

Other management architectures (such as "distributed", "hierarchical", "general network" architectures as described in [10]) containing several managers and agents with multiple detection paths can be modeled and analyzed.

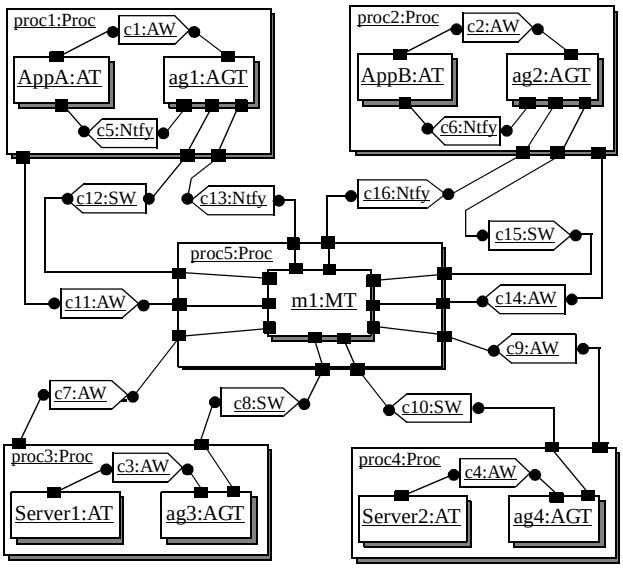

Figure 4. MAMA Model of a centralized management architecture for the system in Figure 1.

4. DETECTION AND RECONFIGURATION

The detection and reconfiguration parameters to be provided in the model are as follows:

- delay of detection propagation from one component to another in the fault management architecture, i.e. a delay parameter associated to each connector in the management architecture. It can be computed from the heartbeat or polling interval for alive-watch and status-watch connectors or from the notification delay for notify connectors.
- delay required by a management component for analyzing and forwarding data.
- restart delay of each application task.
- reconfiguration delay for each service request that has alternative targets.
- probability of successful local recovery of an application task, within a given time interval.

5. MODEL SOLUTION

Let us define a *system state* to be a vector of the states of the fault management components and the components in the software architecture.

The dependability measures for the layered model are then obtained as follows:

1. Construct a continuous-time Markov chain that describes the system changes due to failure and repair and includes the reachable set of system states. It incorporates the detection and recovery behavior of the system in between every two system states.
2. Associate the reward rate equal to 1 to each state of the resulting Markov chain that represents a "working" configuration of the system. Otherwise associate a reward rate of zero with the state. The importance of the fault management architecture is that its failures can modify the system's ability to reach "working" states.
3. Solve the resulting Markov reward model to obtain the desired measures. For example, we can obtain the steady-state availability of the system by summing up the probabilities of all the states that has reward rate equal to 1.

Other interesting measures might be the mean throughput of the system, mean response time for a client, mean outage time for a client of the system etc.

Solvers for these (and more general) measures are presently being developed.

6. CONCLUSION

An approach to incorporate the effect of fault management architectures, that does both failure detection and reconfiguration, in the dependability evaluation of layered systems has been considered. The value of including the management architecture in the analysis is first to account for failures and repairs of managers and agents, and second to evaluate limitations in the fault management architecture.

Current work is to develop a model for capturing the effect of failures and repairs of the management subnet on system dependability measures. The key question to be answered is the complexity of the solution to determine the state probabilities.

7. REFERENCES

[1] Das, O., and Woodside, C.M. The Fault-tolerant layered queueing network model for performability of

distributed systems. IEEE International Computer Performance and Dependability Symposium (IPDS'98), Sept. 1998, pp. 132-141.

[2] Das, O., and Woodside, C.M. Evaluating layered distributed software systems with fault-tolerant features. Performance Evaluation, 45 (1), May 2001, pp. 57-76.

[3] Das, O., and Woodside, C.M. Failure detection and recovery modelling for multi-layered service systems. Fifth International Workshop on Performability Modeling of Computer and Communication Systems, Erlangen, Germany, Sept. 2001, pp. 131-135.

[4] Gokhale, S.S., Wong, W. E., Trivedi, K. S. and Horgan, J. R. An analytical approach to architecture-based software reliability prediction. IEEE International Computer Performance and Dependability Symposium (IPDS'98), Sept. 1998, pp. 13-22.

[5] Goseva-Popstojanova, K. and Trivedi, K. S. Architecture-based approach to reliability assessment of software systems. Performance Evaluation, 45 (2-3), 2001, pp. 179-204.

[6] Hofmeister, C., Nord, R., and Soni, D. Applied Software Architecture. Chapter 4, Addison-Wesley, 2000.

[7] Kreger, H. Java management extensions for application management. IBM Systems Journal, 40(1), 2001, pp. 104-129.

[8] Moser, L.E., Melliar-Smith, P.M., and Narasimhan, P. A fault tolerance framework for CORBA. Proc. of 29th Annual Int. Symposium on Fault-Tolerant Computing, 1998, pp. 150-157.

[9] Schneider, F.B. What good are models and what models are good. Sape Mullender, Editor, Distributed Systems, ACM Press, 1993.

[10] Stamatelopoulos, F., Roussopoulos, N. and Maglaris, B. Using a DBMS for hierarchical network management. Engineer Conference, NETWORLD+INTEROP'95, March 1995.

[11] Tivoli Systems Inc., 9442 Capital of Texas Highway North, Arboretum Plaza One, Austin, Texas. See http://www.tivoli.com.

[12] Woodside, C.M. Performability modelling for multi-layered service systems. Third International Workshop on Performability Modeling of Computer and Communication Systems, Bloomingdale, Illinois, Sept. 1996.