A Distributed State Monitoring Service for Adaptive Application Management♦

Paul Murray
Internet Systems and Storage Laboratory
HP Laboratories Bristol
HPL-2005-73
May 11, 2005*

discovery, distributed state monitoring, resource management, life cycle coordination, failure management

Anubis is a simple state monitoring service that supports coordinated action among distributed management agents. It uses a temporal consistency model that addresses symmetric and asymmetric network partitions. We have used Anubis to support distributed management of adaptive applications in Grid and Utility computing environments and our experience has shown that the abstraction and properties provided by the service simplify the task of programming distributed management behavior. We support this claim by examining three common use cases that our developers encountered, namely: resource management, lifecycle coordination, and compositional failure management.

* Internal Accession Date Only
♦DSN-2005 International Conference on Dependable Systems and Networks, 28 June – 1 July 2005, Yokohama, Japan
Approved for External Publication

A Distributed State Monitoring Service for Adaptive Application Management

Paul Murray
Hewlett-Packard Laboratories
pmurray@hp.com

Abstract

Anubis is a simple state monitoring service that supports coordinated action among distributed management agents. It uses a temporal consistency model that addresses symmetric and asymmetric network partitions. We have used Anubis to support distributed management of adaptive applications in Grid and Utility computing environments and our experience has shown that the abstraction and properties provided by the service simplify the task of programming distributed management behavior. We support this claim by examining three common use cases that our developers encountered, namely: resource management, lifecycle coordination, and compositional failure management.

1. Introduction

The purpose of an application management system is to configure and maintain applications that run in some computing infrastructure. In Grid and Utility computing [7][9][15] the infrastructure is shared among users, and their applications are given access to it as needed. In this context we are interested in distributed applications that are programmed to be reconfigured on the fly, so that they can be adapted to the available pool of resources, recover from failures, or scale to accommodate changing workloads, with management behavior that automatically determines when and how to do this reconfiguration. We call these *adaptive applications*.

The management behavior associated with adaptive applications can be quite complex and application specific. A centralized control point can simplify these issues, but leaves the system exposed to failure of the control point and network partitioning. Another alternative is to fully distribute the control function. This approach is exposed to the complexity of distributed coordination problems. It is this second approach that we address in this paper.

We believe that distributed, adaptive application management can be simplified by the use of a fully distributed system service that provides discovery, monitoring, and failure detection under a single state based abstraction. Here we describe such a service called Anubis. Anubis has a temporal consistency model that supports distributed coordination with properties that span network partitions, allowing even disconnected managers to coordinate their actions. We have developed a fully distributed implementation of our service that has been used in prototype and production systems as a basis for adaptive application management. This experience has highlighted several design patterns that demonstrate the usefulness of the service.

The next section gives an overview of our application management environment. Section 3 describes the Anubis state monitoring service and its timed consistency model, an implementation of which is described in Section 4. Section 5 examines the design patterns identified by users of the service. In section 6 we briefly comment on our experience regarding timeliness and we conclude in Section 7.

2. Overview of Application Management

We base our discussion here around an open source application management framework called SmartFrog (for Smart Framework for Object Groups) developed at HP Laboratories [14]. SmartFrog is a component-based framework that views distributed software systems as collections of cooperating components. It includes a component model, a language for describing collections of components and their configurations, and a runtime environment that activates and manages the described components.

A SmartFrog component implements a set of interfaces that provide management actions, such as creation, activation, and termination, and inspection capabilities such as obtaining the value of attributes associated with the component. A component that has

been created in the runtime environment represents itself and continues to be an accessible managed entity until it is terminated.

SmartFrog components can manage other components and a common configuration is hierarchical composition, in which one component takes responsibility for a group of components (its children) and maps its management actions onto theirs. As an example, a group of web server components could be represented collectively by a single server pool component. The action of instantiating the server pool then leads to instantiation of the web servers. Its termination leads to their termination. This pattern can ultimately be used to manage an entire distributed application as a single component.

In addition to directed action, a component could also implement autonomous action: action that it chooses to take internally. For example, the server pool above could be implemented to maintain the minimal pool required to service ongoing demand.

SmartFrog is implemented in Java as a collection of components that provide a standard behavior. New components with new management functionality, such as the server pool, are implemented by extending these standard components.

3. Anubis State Monitoring Service

A component that implements adaptive management generally bases its behavior on a view of its environment, often provided collectively by discovery, status monitoring, and failure detection services. We developed the Anubis service to provide these under a simple state-based interface suited to adaptive management in the SmartFrog framework. Anubis is a fully distributed, self-organizing, state dissemination service that provides the ability to discover objects, monitor their states, and detect their failure or absence.

3.1. Providers, Listeners, and States

We introduce the following terminology: a *state* is an arbitrary value; a *provider* is a named object that has a state that may change over time; a *listener* is a named object that observes the states of providers. It is the job of the Anubis service to distribute provider states to matching listeners and we say that a listener *observes* the state of a provider when it has a copy of its state value. We do not require that providers have unique names, so a single listener contains a collection of states, each representing a matching provider.

Providers and listeners are registered with the Anubis service interface. Discovery is represented by a listener obtaining the state of a provider. Ongoing observation of the provider's states represents status monitoring. Lack of a state represents failure or absence of the provider. Anubis ensures that listeners observe current provider states in a timely manner.

Our notion of state observation is distinct from the more general publish-subscribe event notification (e.g. TIB/Rendezvous [13]) in that it explicitly addresses *reliable* observation of *current* states. The properties of state dissemination in Anubis are tied tightly to these concepts and target the requirements of adaptive application management.

3.2. Approach

To achieve distributed autonomous management, components need to coordinate their actions based on the information they obtain. So we require a notion of distributed consistency. Moreover, one failure scenario that we wish to deal with is a network partition. When a network partition occurs the components may wish to coordinate adaptive actions, but due to delays in communication and possible asymmetry in the communication paths they may have different views of the state of the system. So we require a consistency model that deals with these conditions.

In many cases a component will use a distributed condition, requiring a comparison of multiple, independently reported states, as the trigger for some action. Typically these require an understanding of concurrency among distributed states. Our model is limited by the fact that component interactions at the application level are not visible or understandable to the management part of the components (e.g. the SmartFrog component may be wrapper code for a legacy application). It is not possible to determine a causal order among component states, so we need an alternative view of concurrency.

We chose to adopt a temporal model based on timed communication paths. This approach is similar to the fail-aware timed model described in [4] and [5] and uses a form of group membership as the basis for temporal consistency. Using group membership as the basis for consistency is common; for example, it is used to attain view synchrony in ISIS [3] and Totem [1]. Fail-aware group membership and communication are described in [6] and [11]. Our protocol is described in [12] and provides the temporal properties stated below.

3.3. Properties

In our model, providers and listeners reside on processing nodes with approximately synchronized clocks, which are connected by a communication network. (Synchronized clocks are not strictly

necessary, but do simplify the implementation and concurrency model.) We assume that the network generally delivers messages between processing nodes reliably and within a bounded communication delay, but may suffer omission or timing failures between given nodes or sets of nodes. Herein we use δ to represent the communication delay bound. We assume that the drift rate of a clock is bounded, but it may fail by drifting excessively. Also a processing node may fail by halting or failing to process messages within a time bound. These failures manifest themselves as communication timing failures. A node's *connection set* is the subset of all nodes that are in fail-free communication with it. A node is always in its own connection set, but otherwise it may change over time.

We say that a node is *stable* when its connection set agrees with those of the nodes in its connection set and has done so for δ time units. Stability indicates transitivity in communication (all these nodes have been in mutual fail-free communication).

Each node derives a *partition view*, which is a modified view of connectivity with the following property:

$$stable_p(t) \Rightarrow \forall q \in System \bullet \\ partition_q(t) \subseteq partition_p(t) \vee \\ partition_q(t) \cap partition_p(t) = \varnothing$$

Where $partition_p(t)$ is the partition view of node p at time t and $stable_p(t)$ is its stability. These time values indicate real (wall clock) time, not the local clock times. We define $observable_p(t)$ to be the set of providers that reside at a node in $partition_p(t)$ and derive the following property for state observation:

$$stable_p(t) \Rightarrow \forall q \in System \bullet \\ observable_q(t) \subseteq observable_p(t) \vee \\ observable_q(t) \cap observable_p(t) = \varnothing$$

As listeners only observe states in their own partition, any states observed are known to be no more than δ time units out of date. The state observation property gives us a level of consistency that allows one management agent to make inferences about what another can observe, and thus provides a basis to solve a number of coordination problems.

4. Anubis Implementation

The Anubis service is implemented entirely in Java as a peer-group of servers. Here we give a brief outline of the Anubis service implementation. A more detailed examination can be found in [12].

Each server includes a partition manager and a state manager layer, as shown in Figure 1 below. The partition manager uses UDP multicast heartbeats to discover and time communication with other servers and TCP connections to provide timed message-based communication between servers.

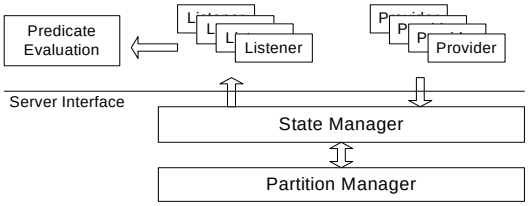

Figure 1. Anubis server architecture.

The partition manager determines the server's partition view and stability using the membership protocol described in [12]. This protocol implements our partition consistency property. In addition the partition manager chooses a distinguished partition member using a leader election protocol.

The partition manager provides the state manager layer with partition view, stability, and leader notifications, and offers message based communication with servers in its partition view.

The state manager layer provides state dissemination between providers and listeners within partitions. In our implementation, listeners discover the state of matching providers within 3δ time units when stable and, once discovered, observe state changes within δ time units regardless of stability. The approximately synchronized clocks allow comparison of independent states as described in [10]. We have implemented a predicate evaluation component with a simple predicate language.

5. Example Use Cases

Anubis has been used as part of the management system for several prototype and live application systems. In each case the management systems were responsible for deployment, runtime management, and tear down of the applications.

These and other systems revealed a few repeated patterns in the way they used Anubis to support the management components. Here we describe three of the design patterns identified, namely resource management, lifecycle coordination, and compositional failure management. For each pattern we describe the advantage of using Anubis as perceived by the implementers, the pattern of use within Anubis, and which properties were exploited by the pattern.

5.1. Resource Management

Several systems used the resource management pattern, including the HP Utility Rendering Service [8]

and (in a simplified form) the GridWeaver print service [2]. This pattern deals with dynamic, on-demand allocation of resources to applications. The resource management pattern is depicted in Figures 4 and 5 below.

The pattern consists of resource components, a resource manager, and an application manager. In the depicted example the resources are machines, the resource manager is responsible for allocating machines to application instances, and the application manager is responsible for assessing and providing for the requirements of an application (e.g. maintain the least number of rendering engines required to service ongoing demand).

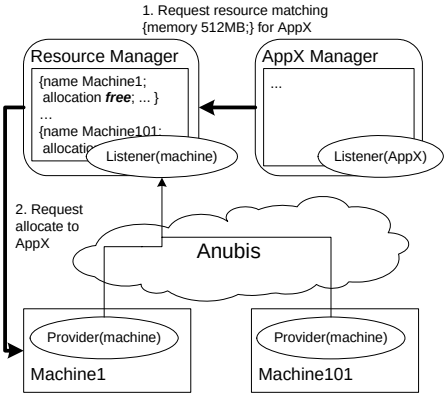

Figure 4. Resource allocation request

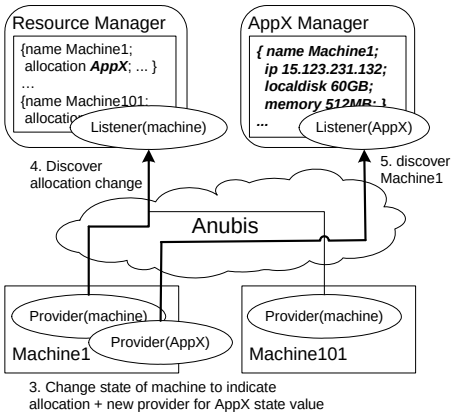

Figure 5. Resource allocation discovery

Each resource registers a provider under the name "machine" with a state value that describes it and its allocation status. The resource manager observes the resources by registering a corresponding listener. The application manager registers a listener under a name assigned to the application (here "AppX"). Typically, the resource manager also observes the application manager using another listener-provider pair, but this is not shown.

If no resources have been allocated the resource manager will observe a pool of resources that report the "free" allocation status, as shown in Figure 4. When the application manager requires a machine it issues a request to the resource manager, which selects an appropriate candidate and passes a request on to it (steps 1 and 2 in Figure 4).

The machine changes its allocation status and registers a new provider under the name "AppX". The change in allocation status is discovered by the resource manager and the machine itself is discovered by the application manager (steps 3, 4 and 5 in Figure 5).

The whole path is optimistic: the resource manager does not guarantee allocation of a resource; if no machine appears the application manager will eventually issue a new request. Resources are de-allocated by the application manager or the resource manager issuing a de-allocate request to the resource (not shown). The change in status is again discovered by all parties concerned.

Advantages: The abstraction provided by the Anubis service unifies many failure cases that do not need to be addressed explicitly. If a machine fails during allocation the application manager will not discover it. If it fails after allocation the application manager will see its state disappear. If the resource manager decides to abruptly de-allocate the machine, again the application manager will see its state disappear. In each case it eventually submits a new allocation request.

Additionally, the machines own their allocation status, not the resource manager, so the status shares fate with the resource. If a resource manager fails it is replaced with no concern for loss of allocation status. The arrangement also tolerates coexistence of multiple resource managers. So again, programming the failure case is simplified.

Pattern of use: Potentially all machines in the system will be communicating state information with the resource manager server. These relationships are long lived. The application manager has a similar but lesser pattern as it manages a (potentially small) subset of the resources.

Properties used: The resource manager exploits the Anubis consistency guarantees to ensure that no two applications believe they own a resource. If the resource manager is stable and can observe a resource, but can not observe the application manager to which it is allocated, it may infer that the resource and the application manager are aware they are partitioned from each other. This is because listeners in stable partitions observe identical or disjoint sets of states. So the state consistency properties guarantee that it is safe

to allocate this resource to another application. As another example, if the resource manager chooses to abruptly de-allocate a machine, if it waits for δ time units before reallocating the machine, all components observing the resource are guaranteed to observe its retraction (regardless of stability).

5.2. Lifecycle Coordination

A problem in all our systems was the need to coordinate application startup. For example, our web server demonstrator required that a database and a logging system be up and running before the web servers could be started. This type of dependency generalizes to any step in the lifecycle of a component.

To coordinate lifecycles each component registers a provider that defines its lifecycle state (deployed, starting, running, terminating, etc.). Lifecycle dependencies among components can be expressed as distributed predicates over these states, so predicate evaluation components can be used to initiate lifecycle transitions. In this pattern all components can be deployed immediately and left to start themselves in an appropriate order.

One notable extension to this pattern is automated application testing. A test management system can deploy an application under test and use predicate evaluation components to determine success or failure of the test; initiating tear-down before deploying the next test.

Advantages and pattern of use: This pattern greatly simplifies programming some behavior, but care needs to be taken to ensure circular dependencies are avoided. Generally the pattern introduces a relatively sparse set of short lived relationships. In the case of ordered start-up a component can stop listening once it has performed the guarded transition.

Properties used: The lifecycle coordination pattern generally requires just reliable delivery of state information. However, any dependency on non-existence of a component can only be determined using stable partition view properties.

5.3. Compositional Failure Management

The example systems all used a composition model promoted by SmartFrog that lead naturally to a pattern for managing failure recovery. Common patterns for implementing fault-tolerance or high availability can be mapped to the composition model. For example, active replication uses a collection of application components to provide redundant processing. Such a group can be managed by a single parent component that replaces members when they fail and represents the group as a single, fault-tolerant entity in the management system. Crash restart uses the ability to recover an application component by restarting it, a role that can be adopted by a parent component. Failure recovery patterns like these were used extensively in the HP Utility Rendering Service.

These patterns can be used to compose an entirely recoverable distributed application. Here we concentrate on hierarchical crash restart. Figure 6 below shows a hierarchy of components that manage a distributed application. The hierarchy needs to remain connected to effectively manage the application, but a node failure or network partition could lead to loss of part of the hierarchy, as represented by the grey nodes.

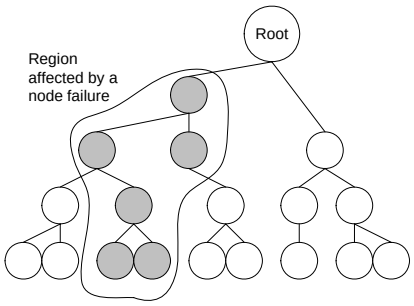

Figure 6. Effect of node failure on a component composition hierarchy

To attack this particular problem a crash-restart component was implemented that used Anubis to implement the following policy:

1. Initially look for children and only create them if they do not already exist.
2. If at any time a child fails then recreate it.

When each component in the hierarchy adopts this crash-restart policy the result automatically fills holes in the tree structure by recursively replacing missing components. The exception is the root of the tree as it has no parent to manage it, so a non-hierarchical approach is required. For this we use an active-standby policy implemented by multiple components that observe each other and deterministically decide which should play the root role at any given time.

Advantages: Anubis provides the ability to determine component existence in stable partitions, so the act of observing a descendant in the hierarchy provides all that is required to determine its existence, locate it using information encoded in its state, and determine its loss in the event of failure or network partitions. Implementing this pattern is trivial.

Pattern of use: As shown in this example, the compositional recovery pattern introduces relationships between components that are typically many-to-one (children-to-parent) and grouped around the components implementing the recovery management

policy (the parent). Otherwise the relationships were relatively sparse.

Properties used: The pattern described relies on the stable partition properties to determine non-existence on start-up. Once past this decision point, the components rely on the consistency properties available in any partition (regardless of stability) to provide failure detection guarantees. Although Anubis is partitionable, most of the applications were not, and in some cases the management system would need to identify a privileged component or a primary partition in order to correctly terminate unwanted components. Anubis does not provide identification of a primary partition directly, but policies based on the ability to observe a given component (e.g. the root component) or the majority of a given set of components, are supported and are generally more applicable.

6. Notes on Timeliness

Protocols based on timeouts are sensitive to the responsiveness of the underlying system. Our service was implemented in Java and ran on non-real time operating systems such as Linux and Windows. However, the timeouts used in practice were generally quite large (the actual value of δ ranged from two to 10 seconds). Neither the JVM nor OS caused unnecessary delays, even in our larger systems (the HP Utility Rendering Service ran across 120 machines and we have tested a system with over 1000 Anubis servers spread over the same 120 physical machines).

One cause of instability we did encounter was misconfiguration of other system services. In one case NTP was configured to use a time source that became inaccessible when a firewall was installed. In another, an NFS server caused the OS on all machines to pause when too many tried to mount file systems simultaneously. In a third, two network switches with incompatible configuration for multicast packet routing overloaded the network. Our partition protocols were sensitive to these issues, but interestingly also proved very useful in debugging the problems.

These cases highlight the importance of correct total system configuration in time-critical systems.

7. Conclusion

We have described how Anubis, a state monitoring service, has been used as part of a distributed management framework for adaptive applications in Grid and Utility computing environments. The use cases presented demonstrate how developers have exploited the service in practice to implement resource management, lifecycle coordination, and compositional failure management. The main advantage of Anubis has been that it hides the complexity of coordinating distributed management agents, simplifying their programming. The state observation abstraction and temporal properties Anubis provides have proven useful in our experience.

8. References

[1] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, P. Ciarfella, "The Totem Single-Ring Ordering and Membership Protocol", *ACM Transactions on Computer Systems*, vol. 13(4), Nov. 1995, pp.311-342

[2] P. Anderson, P. Goldsack, J. Paterson, "SmartFrog meets LCFG - Autonomous Reconfiguration with Central Policy Control", *Proceedings of the 2003 Large Installations Systems Administration (LISA) Conference*, Oct. 2003

[3] K. P. Birman, T. A. Joseph, "Reliable Communication in the Presence of Failures", *ACM Transactions on Computer Systems*, vol. 5(1), Feb. 1987, pp.47-76

[4] F. Cristian, C. Fetzer, "The Timed Asynchronous Distributed System Model", *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing*, 1998

[5] C. Fetzer, F. Cristian, "Fail-Awareness in Timed Asynchronous Systems", *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, May 1996, pp.314-321

[6] C. Fetzer, F. Cristian, "A Fail-Aware Membership Service", *Proceedings of the Sixteenth Symposium on Reliable Distributed Systems*, Oct. 1997, pp.157-164

[7] I. Foster, C. Kesselman, J. Nick, S. Tuecke, "Grid Services for Distributed System Integration", *Computer*, vol. 35(6), June 2002, pp.37-46

[8] Hewlett-Packard, *Servicing the Animation Industry: HP's Utility Rendering Service Provides On-Demand Computing Resources*, http://www.hpl.hp.com/SE3D, 2004

[9] J. O. Kephart, D. M. "Chess, The vision of autonomic computing", *Computer*, vol. 36(1), Jan. 2003, pp.41-50

[10] J. Mayo, P. Kearns, "Global Predicates in Rough Real Time", *Proceedings. Seventh IEEE Symposium on Parallel and Distributed Processing*, Oct. 1995, pp.17-24

[11] S. Mishra, C. Fetzer, F. Cristian, "The Timewheel Group Communication System", *IEEE Transactions on Computers*, vol. 58(8), Aug. 2002, pp.883-899

[12] P. Murray, "The Anubis Service", *Hewlett-Packard Laboratories Technical Report*, HPL-2005-72, 2005

[13] B. Oki, M. Pfluegl, A. Siegel, D. Skeen, "The Information Bus – An Architecture for Extensible Distributed Systems", *Proceedings of the 14th ACM Symposium on Operating System Principles*, Dec. 1993, pp.58-68

[14] SmartFrog Reference Manual v3.02, http://www.smartfrog.org/, July 2004

[15] J. Wilkes, J. Mogul, J. Suermondt, "Utilification", *Proceedings of the 11th ACM SIGOPS European Workshop*, Sept. 2004