Anubis User Guide

Paul Murray

For use with version 1.8
Last Update: 12th February 2007

1 Introduction

Anubis is a notification service for providing accurate, time-bound detection of distributed system status. Examples include appearance or start-up of system components, disappearance or termination of system components, and the occurrence of specific component states or combination of distributed state. The service is intended to support distributed, reactive system management.

This document is aimed at programmers who intended to use the Anubis Service with SmartFrog [1]. The document contains simple examples that exemplify the interface. See [2] for examples of how Anubis has been used in complex systems to support capabilities such as distributed resource management and failure recovery and [3] for details of the Anubis service protocols and their properties.

2 State Detection Model

Anubis is a service that provides the ability to observe states across a distributed system. It passes state information from an observed entity to an observing entity within a guaranteed time bound. For convenience we call the observed entity the state provider (or just the provider) and we call the observing entity the state listener (or just the listener).

Each provider has a value that we call its state. At any moment in time it will be in a given state, determined by its value, and over a period of time it may transition through a succession of states, i.e. the value may change. The challenge for the Anubis service is to tell the listener the current state of the provider.

At the provider, what we mean by *current state* is clear: it is its value at the present time. However, although Anubis communicates the state of the provider to the listener within a time bound, there is some limited variability in when the listener observes a new state. So at the listener, the current state is actually the last state that the listener observed. We call the last state observed by the listener the *detected state* or its *detection*. So, informally, a detection is the current state of a provider as determined by a listener.

We also introduce the notion of observational-stability (or *o-stability*). A state becomes o-stable when it does not change for a period of time (in fact the same known time bound for which a detection is

guaranteed). O-stable states are interesting in that they can be compared with respect to time. The variability in the delivery of a state observation to the listener means that multiple detections that coexist (e.g. detections of multiple listeners) cannot be said to represent states that coexisted. However, detections of o-stable states do represent states that coexisted. This property supports capabilities listed below.

Providers and listeners associate themselves with a name. The Anubis service communicates the states of providers to those listeners that are associated with the same name. Each provider or listener can associate itself with a just one name; however, multiple providers can use the same name and likewise for listeners. Each listener is told about the state of every provider associated with its same name. Although the listeners can distinguish between providers, the only thing they know about them is the state information. A single Anubis system has a single, global name space.

The detailed semantic model of state detection are beyond the scope of this document. Detection is a very powerful concept and supports the following useful capabilities:

- **Time-bound Detection** – a listener will discover the existence of a provider and its current state within a known period of time (say T). If the provider exists in the system prior to the listener, the listener will discover the provider and its state no more than time T after the listener is created. If the listener exists first then the listener will discover the provider and its state no more than time T after the provider is created. When the provider and listener already exist the listener will detect changes in the provider's state no more than time T after they happen.

- **Non-existence Detection** – if a provider vanishes from the system (e.g. it silently terminates), the provider will detect its disappearance after a known time bound. Also, if a listener is created and there is no provider for it to observe, it will know that no such provider exists after the same time bound.

- **Distributed Predicate Evaluation** – the states observed by listeners can be correlated with respect to the time if they are o-stable. This can be used to infer distributed state observations from multiple detections.

- **Group Membership** – a listener detects the existence and state of all providers that use the same name. So the listener implicitly knows the collection of providers, and hence Anubis implements group membership. Groups can be o-stable (as well as the individual states) and multiple listeners observe o-stable group membership changes consistently. This group interpretation can be viewed as a special case of distributed predicate evaluation.

- **Mutually Concurrent Agreement** – two different listeners listening to the same providers will detect the same states within a known time bound of each other; a property called mutually concurrent agreement. This property allows multiple agents to take consistent actions, based on detections, without communicating with each other, and is a basis for scalable, reactive management. Mutually concurrent agreement extends to distributed predicate evaluation.

Although the Anubis service is capable of supporting all these properties, the interfaces described in this document only directly cover time-bound detection and non-existence detection. Group membership is covered, but only the weaker variant that does not deal with o-stability and distributed consistency of group views. The rest of the properties can be implemented on top of the Anubis service interfaces that are described in this document, including the stronger o-stability related properties.

3 Partitions

Communication partitions form the basis of the strong semantic guarantees that Anubis provides. Each Anubis server attempts to identify a collection of peer-servers that have the mutual property of being able to communicate in a bound period of time. Such a collection is called a *partition* and the job of a server is to identify the partition that contains it. The smallest partition a monitor can be in contains only itself.

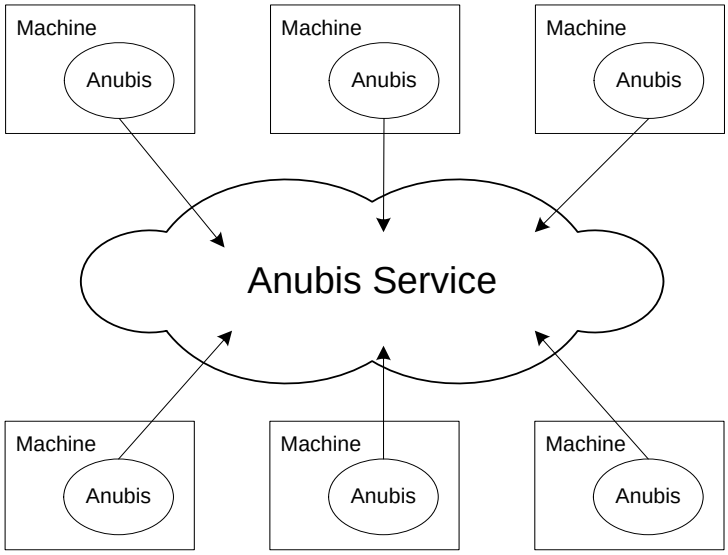
The partition protocol guarantees the ability to communicate within a partition partition. If there are Anubis servers in the system that cannot communicate with each other in the given time bound they will fall into separate partitions. Every server in the partition will recognise this fact, so two partitions are guaranteed not to overlap.

We say that the Anubis service is *partitionable* because each server can identify its own partition and therefore recognise its own part of a system if it fragments due to communication problems or failures. If the communication problems are corrected the servers will rediscover the entire system.

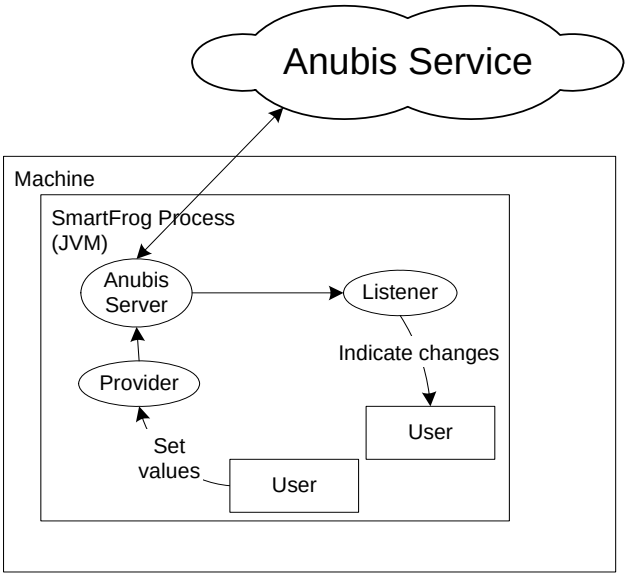
Listeners can only detect the state of providers that are in the same partition as them. If a provider is separated from a listener due to a partition change, the listener will be informed it has disappeared. If the user wishes he can interpret the absences as failure. If so then the Anubis service non-existence detection can be interpreted as time bound failure detection. Anubis itself makes no such distinction.

4 Use Model

The Anubis service consists of a number of server components that interact with provider and listener components. The provider and listener components form the user interface to the service. The server components implement the service. This is depicted in the diagram below.



Each Anubis server is located on a separate machine and provides access to the service on from that machine. The service is always accessed using local interfaces, so an instance of the server component must reside on each machine to be covered by the service. These servers find each other and form a peer group. In fact the protocols that determine the peer group are the protocols that guarantee the time bound communication of the service. The suggested limit of one Anubis server component per node is suggested to limit the processing overhead on the machine. It is entirely possible to use more than one server on each machine, but no advantage is gained by such an arrangement.



The user interface to the service is implemented in the form of provider and listener components that are registered with the local server, as shown in the diagram above. The provider and listener components are abstract classes that the user specializes. The provider takes the role described in the state detection model above. It has a value and is known to the Anubis service. The user can set the value of the provider, thus implementing a push model to make user defined state values visible in the Anubis service.

The listener component also takes the role described in the state detection model above. It is registered with the local Anubis server and provides an asynchronous up-call interface to the user. The up-calls notify the user of detections, thus implementing a push model delivering for delivery of state values to the user. The listener also provides a collection view interface that allows the user to iterate over the state values known to the listener. This implements a pull model interface for the user to browse detections.

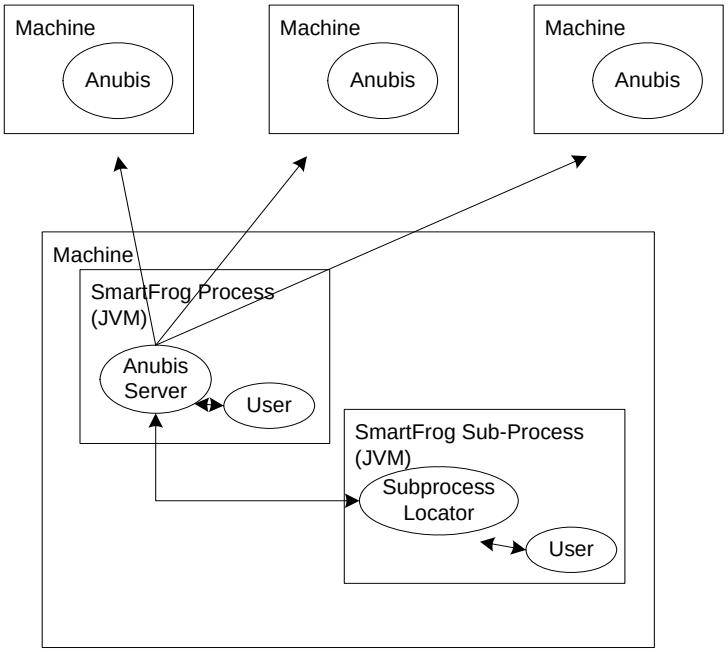
5 Anubis Servers

The Anubis service consists of a collection of Anubis Server components. The Anubis Server component provides local access only; so to access the service a Server component must be running in the same JVM as the interface user. This is deliberate and is intended to support common fate between the server and the user.

It is entirely possible to run more than one Anubis Server in a single JVM (even two that are in the same Anubis system), although there is an unnecessary performance overhead for doing so. The

intention is that only one Anubis Server would run per machine, however to more efficiently support sub-processes (in SmartFrog terminology), we provide a *sub-process locator*: essentially a proxy interface for the server that resides in another JVM on the same machine. Where there are sub-process it is usual to deploy the Anubis Server component in the root process.

The arrangement of one Anubis Server per machine, with subsequent sub-processes having sub-process adapters is shown in the diagram below.



The Anubis server component may be deployed independently of any user components or it could be incorporated in the same description as the user. Either way the user component will need to find the service interface to use it. First we describe how to start the Server and Sub-Process adapter.

All the template descriptions for the Anubis service are contained in the SmartFrog template file:

`/org/smartfrom/services/anubis/components.sf`

The relevant descriptions for the Anubis server and sub-process locator are named:

`AnubisService`

`AnubisSubProcessLocator`

5.1.1 AnubisService

The AnubisService component should be specialised by extending its anubisParameters attribute. The value of this attribute should be a block of attributes as defined in the SmartFrog component DefaultParameters – over ridden as appropriate. The parameters are as follows:

identity:node an integer identifier for this server instance. This should be a low non-negative number. It used by the partition protocol and should be unique in the Anubis system.

identity:magic The magic number is used in the partition protocol to avoid conflicts when more than one Anubis service is in use in the same network. The default is 123456.

heartbeatGroup:host This should be a multicast group IP address in string form. The address should be the same for every server in a single Anubis system. The default is "233.1.2.30". Multiple Anubis systems can be constructed that will not interact with each other by giving each system a different multicast address.

heartbeatGroup:port This is the port number for the multicast group. Similar conditions apply as for the address. The default is 1966;

heartbeatInterval The heartbeat interval is the first of two parameters that define the timing semantics of the Anubis system. It defines how freaquently heartbeats are sent in the time checking protocol. A server is considered timely if it can communicate messages with other servers within a time bound defined by heartbeatInterval*heartbeatTimeout. This parameter is a positive long number representing milliseconds between heartbeats. The default is 2000.

heartbeatTimeout The heartbeatTimout is the second parameter that defines the timing semantics. It is a positive integer that represents the maximum number of heartbeat intervals that a server can delay communicating messages before it is considered untimely. The default is 3.

heartbeatTransport The communication transport layer is composed of two pluggable components (one for multicast communication and one for connection oriented communication). Currently there is only one version of the multicast heartbeat transport, defined by extendeding this attribute with MulticastHeartbeatTransport. Future alternatives would be specified here.

connectionTransport Currently there are two versions of the connection oriented part, one uses blocking communications with a thread per connection, and one uses non-blocking communication serviced by a single shared thread pool. This attribute should extend either BlockingConnectionTransport or NonBlockingConnectionTransport to select the desire version. As of version 1.5 these transports are interoperable; the default is NonBlockingConnectionTransport.

heartbeatProtocolFactory There are currently two alternative heartbeat protocols that can be specified: the timed protocol and the ping protocol. The timed protocol requires synchronised clocks across the group and provides a partition wide time reference. The ping protocol does not require synchronised clocks, but as a consequence can not provide a partition wide time reference. The protocol is selected by extended this attribute with TimedHeartbeatProtocol or PingHeartbeatProtocol as required.

leaderProtocolFactory There is currently only one leader election protocol available. This is specified here by extending this attribute with MajorityLeaderProtocol.

testable There is a monitoring tool (described below) that allows an Anubis system to be observed and debugged. If the value of this attribute is "true" the server will interact with the tool to provide some statistics and details of its protocol states. If the value is “false” then the server will not interact with the monitoring tool and the only information available will be that which can be gathered by snooping on protocol message exchanges.

preferred The partition membership protocol identifies a *leader* node – that is, it designates a unique node in the partition that is recognised by all members. By default all nodes have equal chance of becoming the leader, but certain nodes can be identified as *preferred* nodes by setting this attribute to true. The partition will always select a preferred node as the leader if one exists in the group. Leadership is not currently visible through the Anubis Service interface.

One further parameter can be specified. This is not contained in the anubisParameters attribute. This attribute is anubisServiceName and is described below:

anubisServiceName The server will register itself under the name given as the string value of this attribute, in the local process compound (root SmartFrog component for this JVM). If this name is specified it can be used to locate the server’s interface. If not, other means will need to be used, such as traversing the deployment tree.

As a minimum an AnubisService component should be given an indentifier attribute that is unique in the system. However it is advisable to modify the magic number, multicast address and multicast port, to avoid unintended interaction with other Anubis systems in the same network.

An example SmartFrog description that will deploy a an Anubis Server component with the node identifier 1, magic number 1234, and port number 32323 on a machine named murray-p.hpl.hp.com is as follows (note that sfProcessHost is a standard SmartFrog attribute that determines where to deploy a component):

```
config extends AnubisService {
      sfProcessHost    "murray-p.hpl.hp.com";
      anubisParameters extends DefaultParameters {
            node             1;
            identity:magic   1234;
            heartbeatGroup:port 32323;
      }
}
```

The AnubisSubProcessLocator template contains only two attributes:

anubisServiceName The sub-process locator component will register itself under the name given as the string value of this attribute, in the local process compound (root SmartFrog component for this JVM). If this name is specified it can be used to locate the server's interface. If not, other means will need to be used, such as traversing the deployment tree.

anubisParameters The same set of parameters given above for AnubisService. Many of these parameters are not pertinent to the AnubisSubProcessLocator, but it is convenient to use the same set.

anubisAdapter A reference to a remote interface adapter in the Anubis Service component. The Anubis Service component is actually deployed with a remote interface adapter component. This attribute should be a lazy reference to it. The default value for the attribute is:
LAZY HOST localhost:anubis:subProcessAdapter. This assumes that the Anubis Server process has been deployed in the root process and has the name "Anubis". If this is not the case the attribute should be modified to reflect the appropriate location.

An example SmartFrog description that will deploy a server and create a sub-process with a sub-prcess locator component in it, with attributes similar to the AnubisService example above, is as follows (note that sfProcess is a standard SmartFrog attribute that determines a sub process to deploy the component in):

```
config extends DetachingCompound {
      sfProcessHost "murray-p.hpl.hp.com";
      parameters extends DefaultParameters {
            identity:node 1;
            identity:magic 1234;
```

```
            heartbeatGroup:port 32323;
        }
        server extends AnubisService {
            anubisParameters parameters;
        }
        subprocess extends AnubisSubProcessLocator {
            sfProcess “subprocessName”;
            anubisAdapter LAZY ATTRIB:server:subProcessAdapter;
            anubisParameters parameters;
        }
    }
```

5.1.2 Monitoring Tools

An Anubis system can be observed using two tools that were originally intended for debugging Anubis, but have since proved useful in testing user code. They are:

- A partition monitor; a tool that presents a graphical representation of the system, as observed from a single location.
- A locator test component used to test the location service by observing the contents of a single server’s visible information and exercise the locator interface.

These tools are described fully later.

6 Programmatic Interface

The classes that constitute the programmatic interface for Anubis are all held in the com.hp.sfServices.Anubis.Locator package. They are:

```
org.smartfrog.services.anubis.locator.AnubisLocator;

org.smartfrog.services.anubis.locator.AnubisProvider;

org.smartfrog.services.anubis.locator.AnubisListener;

org.smartfrog.services.anubis.locator.AnubisValue;

org.smartfrog.services.anubis.locator.AnubisStability;
```

There is a special note on using Anubis from sub-processes relating to java class loading at the end of this section.

6.1 AnubisLocator

The user can access the Anubis service by obtaining a reference to the AnubisLocator interface. This can be done using the SmartFrog sfFind() method as described above.

The AnubisLocator interface has six methods that are of interest to the user. These are: registerProvider(…), deregisterProvider(…), registerListender(…), deregisterListener(…), registerStability(…), and deregisterStability(…);

The first two, as the names suggest, register and deregister providers respectively. The signatures for these are given below. Each takes an AnubisProvider class instance as its one parameter. This is the provider that is being registered/deregistered. The AnubisProvider class is described below. When a provider has been registered with the Anubis service using registerProvider() it becomes visible to the service and listeners will become aware of its presence as appropriate. When it has been deregistered using deregisterProvider() it will cease to be visible to the service and any listeners that are aware of it will become aware of its absence (i.e. they will recognise that the corresponding value has gone).

```
void registerProvider(AnubisProvider provider);

void deregisterProvider(AnubisProvider provider);
```

The second two register and deregister listeners. The signatures for these are given below. Each takes an AnubisListener class instance as its one parameter. This is the listener that is being registered/deregistered. The AnubisListener class is described below. When a listener is registered with the Anubis service it will become aware of values as appropriate; when it is deregisters it will cease to be informed of any values – its awareness of values becomes undefined.

```
void registerListener(AnubisListener listener);

void deregisterListener(AnubisListener listener);
```

The last two methods register and deregister stability notification objects. The signatures are given below. Each takes an AnubisStability class instance as its one parameter. AnubisStability class instances are used to notify the user of local partition stability and time reference.

The AnubisLocator interface is re-entrant. By this we mean that it is ok to call the interface during an upcall from the Anubis service (i.e. in an AnubisListener upcall – as described below).

6.2 AnubisProvider

The AnubisProvider class is used to provide values to the Anubis service. The user creates a named object by constructing an instance of this class. The class has two basic methods of interest: AnubisProvider() the constructor, and setValue(), which is used to set the current value associated with the class. The signatures for these methods are given below.

```
AnubisProvider(String name);

void setValue(Object obj);
```

One further method in the AnubisProvider class is setMarshallValues() (below). This is an advanced feature of Anubis described later in section 6.5. The method is given below, but we will ignore it at present.

```
static void setMarshallValues(Boolean marshall);
```

When the AnubisProvider class has been constructed its default value will be null. Anubis recognises null as a valid value and differentiates it from the absence of a value. A value is only absent when there is no provider for it.

The only restriction on the value that can be set is that it is a serializable class – i.e. it implements the java.io.Serializable interface. These values will be communicated and hence need to be serialized.

The setValue() method can be called at any time, before or after registering the provider. Listeners may observe the value that the provider holds at the time it is registered with Anubis. If this is not desired the setValue() method should be used to set the value before registering the provider using the AnubisLocator interface.

6.3 AnubisListener

The AnubisListener class is used to observe values in the Anubis service. This is an abstract class because it contains methods that must be defined by the user. So the AnubisListener class must be specialised. The AnubisListener can be thought of as a view in the database sense of the word. It will contain a collection of all the values associated with the name of the listener, one for each provider with that name. The values in the collection are represented by instances of the AnubisValue class, described below. The user will be notified when changes in the collection occur using upcalls. The upcalls are the methods that need to be specialised.

6.3.1 Basic Use

The following basic methods are of interest: AnubisListener() the constructor, newValue(AnubisValue value), which indicates a change in value (either a new provider has appeared or the value provided by an existing one has changed), and removeValue(AnubisValue value), which indicates that a value has disappeared (either because the provider has deregistered or become partitioned – either way it is no longer visible). The signatures of these methods are given below.

```
AnubisListener(String name);

void newValue(AnubisValue value);

void removeValue(AnubisValue value);
```

When the AnubisListener class is constructed it is given a name. This is the name that will be used to identify values in the Anubis service. The name cannot be changed once assigned.

When the AnubisListener class instance is registered via the AnubisLocator interface it will become aware of any values associated with its name that are visible to Anubis. This occurs as follows: for each provider that is visible through Anubis an instance of AnubisValue will be created in the AnubisListener's collection and the newValue() upcall will be invoked, also passing the AnubisValue class as the parameter of the call. A single AnubisValue class instance directly corresponds to a single AnubisProvider instance. If the value associated with a provider changes, the value associated with the corresponding AnubisValue instance changes and the newValue() upcall will be invoked again, referring to the appropriate AnubisValue. If a provider deregisters or disappears, the corresponding instance of AnubisValue will be removed from the AnubisListener's collection and the removeValue() upcall will be invoked, passing the AnubisValue instance as its parameter. The AnubisValue instance will contain the null value.

6.3.2 Specialising the value handling

AnubisValue can be specialised to create a user-defined subclass. The AnubisListener uses a factory method called createValue() to construct instances of AnubisValue. If the AnubisValue class is specialised the user will need to over-ride the definition of createValue() to construct the new subclass instead. The signature of createValue() is given below.

```
AnubisValue createValue(ProviderInstance i);
```

The presence of the ProviderInstance class in the interface is regrettable as it is an internal representation of which the user should not be aware (this is a first pass at the interface!!☺) The AnubisValue class constructor takes this parameter, so the user's constructor should look something like:

```
MyAnubisValue(ProviderInstance i) {
      super(i);
      /* any other code …*/
}
```

A common use pattern for this feature is to turn upcalls on the AnubisListener class into upcalls on instances of the user class related to the values, as in the following example:

```
MyAnubisListener extends AnubisListener {
      MyAnubisListener(String name) { super(name); }
      void createValue(ProviderInstance i) {
            return new MyAnibsValue(i);
      }
      void newValue(AnubisValue value) {
            ((MyAnubisValue)value).newValue();
      }
      void removeValue(AnubisValue value) {
            ((MyAnubisValue)value).removeValue();
      }
}
```

6.3.3 Further convenience methods

There are a few other methods in the AnubisListener class that allow the user's convenience. These are given below:

```
String getName();

long getUpdateTime();

int size();

Collection values();
```

The getName() method returns the name of the listener. The getUpdateTime() method returns the most resent time stamp associated with any value in the collection. Note that the most recent time stamp does not necessarily correspond to the most recent addition or change to the collection. Although Anubis provides guarantees regarding time-bound distribution of information, and ordered updates from a particular provider, it does not provide time ordered updates between multiple providers.

The size() method returns the number of values in the listener's collection. The values() method returns a collection view of the listener's AnubisValue class instances.

6.4 AnubisValue

The AnubisValue class is used as a container for the actual values associated with a listener. The value contained by an AnubisValue instance is a copy of the object that was set for the associated provider. In addition to containing the actual value the AnubisValue class holds a some extra information including the name associated with the value (the same name used by the listener and the provider), the time stamp assigned to the value (the time stamp is set by the provider when the value is set or when the provider registers, and by the local node when the provider disappears), and a globally unique identifier for the provider. The signatures for these methods are given below.

```
String getName();

String getInstance();

long getTime();

Object getValue();
```

The getName() method returns the name, getInstance() returns the global identifier, getTime() returns the time stamp, and getValue() returns the actual value object. Note that when a value has been removed as indicated by the AnubisListener.removeValue() upcall it will contain the null value and the time stamp will reflect the time that the corresponding provider deregistered or the time that the local node determined its disappearance.

6.4.1 Using the unique identifier

The globally unique identifier (or instance) is actually of little use to the user unless he wants to recognise a provider that disappeared (say due to a network partition) and then reappeared. In this eventuality the provider will still have the same unique identifier, although when it reappears the listener will construct a new AnubisValue container for it. This can be contrasted with the case where a provider deregisters and then registers again, in which case Anubis will treat it as a new provider and assign a new globally unique identifier.

6.5 AnubisStability

The AnubisStability class is used to observe the stability of the local Anubis partition. This is an abstract class containing a single method that is used to notify the user of changes in the local partitions stability and time references as follows:

```
void stability(Boolean isStable, long timeRef);
```

The stability method should be over ridden by the user to implement the desired notification. The AnubisStability class uses a default constructor (no parameters).

When the user registers the AnubisStability class with the AnubisLocator interface there will be an initial notification giving the current partition stability and time reference. From then on the user will be informed when the partition changes between stable and unstable, giving the associated time references. Registration and deregistration are asynchronous operations and the notifications (including the initial one) are delivered in separate threads.

6.6 Anubis Sub-Process Interface and Java Class Loading

When using multiple isolated JVMs to run a distributed application it is sometimes suitable to use different class paths or code bases for the different JVMs. This may be for security purposes or

merely convenience. In these cases it is possible that the user may want to set provider state values to classes that are not recognised in all JVMs. When using the Anubis sub-process locator interface it is entirely possible that the user's state value classes will not be recognised by the JVM that contains the Anubis server. To deal with this situation the Anubis service has the option of converting user state values to an internal representation. This uses class marshalling to copy the users state values to a transferable form during the AnubisProvider.setValue(…). The listener class perform the appropriate inverse operation. Anubis can handle these marshalled state values in JVMs that do not recognise the classes, avoiding the need to unnecessarily include user application classes in the class path or code base of the Anubis server's own JVM.

This copying is an extra overhead, so the facility is switched off by default. State value marshalling can be enabled at the provider end by calling the following static method:

```
AnubisProvider.setMarshallValues(true);
```

This call will cause state value marshalling to be used by all providers in the JVM in which it is called.

7 Examples

In this section we give an example use of the Anubis service interfaces to implement a group membership monitor. In this scenario we create a component that observes a collection of components in the system and their status. In this case the status will indicate if they are in the deployed or started states in the SmartFrog life cycle. Their absence will indicate either that they don't exist or that they did, but have terminated or were otherwise removed from the system.

The components that are going to be observed use the provider interface. The AnubisPrim component in Appendix A is the component that will be observed. The component has three simple steps to implement:

1. It obtains a reference to the Anubis server interface.

   ```
   locator = (AnubisLocator)sfFind("locator");
   ```

   We assume here that there is an attribute named "locator" in the component description for the AnubisPrim component that references the interface for the AnubisLocator interface of the server.

2. It creates an instance of the provider class and sets its value.

   ```
   provider = new AnubisProvider(myName);
   provider.setValue("deployed");
   ```

3. It registers the provider class with the Anubis service.

```
locator.registerProvider(provider);
```

These steps are performed during the deploy phase and result in the component state represented by the value “deployed” being visible in the Anubis service under the name held by the variable myName.

During the start phase the value of the provider is set to “started” to indicate its new state. During the termination phase the provider is de-registered from the Anubis service.

The group monitor component uses the listener interface. This involves specializing the AnubisListener (see GroupListener in Appendix B) and AnubisValue (see GroupMember in Appendix C) classes. The actual group monitor component is the SmartFrog component defined in Appendix D.

The GroupListener class is an AnubisListener class that has been specialised to create instances of the GroupMember class instead of the default AnubisValue. This is done by over-riding the createValue() method as follows:

```
public AnubisValue createValue(ProviderInstance providerInstance) {
    return new GroupMember(providerInstance, this);
}
```

We have also implemented the newValue() and removeValue() methods in GroupListener to invoke calls on the instance of GroupMember referred to in the up-call. The purpose here is to convert up-calls on the listener into up-calls on the classes that represent the values themselves. newValue() and removeValue() are defined as:

```
public void newValue(AnubisValue v) {
    ((GroupMember)v).newValue();
    printGroup();
}

public void removeValue(AnubisValue v) {
    ((GroupMember)v).removeValue();
    printGroup();
}

public void printGroup() {
    String str = "Group “ + getName() + “: [ ";
    Iterator iter = values().iterator();
    while( iter.hasNext() ) {
        str += ((AnubisValue)iter.next()).getValue() + " ";
    }
    System.out.println("Group " + getName() + ": " + str + "]");
}
```

The GroupMember class simply implements a version of the AnubisValue class that outputs messages on the output stream to indicate what is happening. The class has the two additional methods called by the GroupListener class.

```
public void newValue() {
    System.out.println("New value in group " + getName() +
                       " = " + getValue());
}

public void removeValue() {
    System.out.println("Removed value from group " + getName() +
                       " = " + getValue());
}
```

When instances of the providers observed by these classes appear, change value or disappear the code above simply outputs messages to indicate what has been observed. For example, if listening to an AnubisPrim being deployed under the name "webServer" the listener would output:

```
New value in group webServer = deployed        (from GroupMember)

Group webServer: [ deployed ]                  (from GroupListener)
```

If another is deployed the listener would output:

```
New value in group webServer = deployed        (from GroupMember)

Group webServer: [ deployed, deployed ]        (from GroupListener)
```

And when one has started:

```
New value in group webServer = started         (from GroupMember)

Group webServer: [ started, deployed ]         (from GroupListener)
```

Note that the instance of the AnubisMember class does not change when it has a new value. The listener does understand that it is the same component that was deployed that is now started. Note also that the values that have been passed are the strings "deployed" and "started". Any serializable objects can be used as values, they do not need to be strings.

Note also that there is a short delay between a new provider registering with Anubis and the listener discovering that it is there. This delay is time-bounded when the system is stable, but it is still there. Usually SmartFrog components are only in the deployed state for a short period of time. When a listener discovers a provider it observes its *current* state, therefore it is entirely possible that the provider is set to "started" before the listener has discovered it. When running these examples, observation of the "deployed" state depends on a race condition between the discovery mechanism

and the AnubisPrim's transition to the "started" state – this is correct behaviour. However, once a provider has been discovered, the listener will observe all of its state transitions, in order, until it is removed from the system.

AnubisMonitor is the SmartFrog component that deploys the listener. Like the AnubisPrim class, it obtains the Anubis server interface, creates the listener and registers the listener:

```
locator   = (AnubisLocator)sfFind("locator");
listener  = new GroupListener(groupName);
locator.registerListener(listener);
```

The only other action taken by AnubisMonitor is to deregister the listener when it terminates.

The last class in our examples is StabilityNotices, a class that uses the stability notification part of the Anubis service interface. StabilityNotices contains an inner class that specialises the AnubisStability class to implement the stability(…) upcall as follows:

```
public void stability(boolean isStable, long timeRef) {
     if( isStable ) {
          System.out.println("****** Partition has " +
               "stablized with time reference " + timeRef );
     } else {
          System.out.println("****** Partition is UNSTABLE");
     }
}
```

The AnubisStability class is registered with the Anubis service interface in the same way as before:

```
locator = (AnubisLocator)sfResolve("locator");
stability = new Stability();
locator.registerStability(stability);
```

When the Anubis stability status changes the interface invokes the stability(…) method to notify the user.

SmartFrog descriptions that accompany these examples can be found in:

```
org/smartfrog/services/anubis/components/examples/components.sf
```

To run the examples use the descriptions of AnubisNode, Monitor and WebServerGroup in:

```
org/smartfrog/services/anubis/components/examples/examples.sf
```

First deploy a copy of AnubisNode to create an Anubis service. Then deploy a copy of Monitor and finally deploy a copy of WebServerGroup. The Monitor will display information about stability and its observations on the standard output stream.

8 References

[1] The SmartFrog Reference Manual – A Guide to Programming with the SmartFrog Framework, http://smartfrog.org

[2] P. Murray, A Distributed State Monitoring Service for Adaptive Application Management, *DSN-2005 International Conference on Dependable Systems and Networks*, June 28th – July 1st 2005, Yokohama, Japan

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

Appendix A AnubisPrim.java

```
/** (C) Copyright 1998-2005 Hewlett-Packard Development Company, LP

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For more information: www.smartfrog.org

*/
package org.smartfrog.services.anubis.components.examples;


import java.rmi.RemoteException;

import org.smartfrog.services.anubis.locator.AnubisLocator;
import org.smartfrog.services.anubis.locator.AnubisProvider;
import org.smartfrog.sfcore.common.SmartFrogException;
import org.smartfrog.sfcore.prim.Prim;
import org.smartfrog.sfcore.prim.PrimImpl;
import org.smartfrog.sfcore.prim.TerminationRecord;
import org.smartfrog.sfcore.reference.Reference;

public class AnubisPrim extends PrimImpl implements Prim {

    static private Reference MARSHALL_REF = new Reference("marshall");
    static private Reference NAME_REF     = new Reference("anubisName");
    static private Reference LOCATOR_REF  = new Reference("locator");

    private AnubisLocator  locator        = null;
    private AnubisProvider provider       = null;
    protected String         myName         = null;

    public AnubisPrim() throws Exception {
            super();
    }

    public void sfDeploy() throws SmartFrogException, RemoteException  {
        try {
            super.sfDeploy();
            myName = sfResolve(NAME_REF, (String)null, false);
            if( myName == null)
                myName = sfCompleteName().toString();

            AnubisProvider.setMarshallValues(sfResolve(MARSHALL_REF,
```

```
                                                        false, false));

            locator = (AnubisLocator)sfResolve(LOCATOR_REF);
            provider = new AnubisProvider(myName);
            provider.setValue("deployed");
            locator.registerProvider(provider);
        }
        catch (Exception ex) {
            ex.printStackTrace();
            throw (SmartFrogException)SmartFrogException.forward(ex);
        }
    }

    public void sfStart() throws SmartFrogException, RemoteException  {
        try {
            super.sfStart();
            provider.setValue("started");
        }
        catch (Exception ex) {
            ex.printStackTrace();
            throw (SmartFrogException)SmartFrogException.forward(ex);
        }
    }

    public void sfTerminateWith(TerminationRecord tr) {
        locator.deregisterProvider(provider);
        super.sfTerminateWith(tr);
    }

}
```

Appendix B GroupListener.java

```
/** (C) Copyright 1998-2005 Hewlett-Packard Development Company, LP

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For more information: www.smartfrog.org

*/
package org.smartfrog.services.anubis.components.examples;


import org.smartfrog.services.anubis.locator.AnubisListener;
import org.smartfrog.services.anubis.locator.AnubisValue;
import org.smartfrog.services.anubis.locator.names.ProviderInstance;
import java.util.*;

public class GroupListener extends AnubisListener {

    public GroupListener(String name) {
            super(name);
    }
    public AnubisValue createValue(ProviderInstance providerInstance) {
        return new GroupMember(providerInstance, this);
    }
    public void newValue(AnubisValue v) {
        ((GroupMember)v).newValue();
        printGroup();
    }
    public void removeValue(AnubisValue v) {
        ((GroupMember)v).removeValue();
        printGroup();
    }
    public void printGroup() {
        String str = "[ ";
        Iterator iter = values().iterator();
        while( iter.hasNext() ) {
            str += ((AnubisValue)iter.next()).getValue() + " ";
        }
        System.out.println("Group " + getName() + ": " + str + "]");
    }
}
```

Appendix C GroupMember.java

```
/** (C) Copyright 1998-2005 Hewlett-Packard Development Company, LP

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For more information: www.smartfrog.org

*/
package org.smartfrog.services.anubis.components.examples;


import org.smartfrog.services.anubis.locator.AnubisValue;
import org.smartfrog.services.anubis.locator.names.ProviderInstance;

public class GroupMember extends AnubisValue {

    GroupListener group;

    public GroupMember(ProviderInstance instance,
                       GroupListener groupListener) {
        super(instance);
        group = groupListener;
    }
    public void newValue() {
        System.out.println("New value in group " + getName() +
                           " = " + getValue());

    }
    public void removeValue() {
        System.out.println("Removed value in group " + getName() +
                           " = " + getValue());
    }

}
```

Appendix D GroupMonitor.java

```
/** (C) Copyright 1998-2005 Hewlett-Packard Development Company, LP

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For more information: www.smartfrog.org

*/
package org.smartfrog.services.anubis.components.examples;


import java.rmi.RemoteException;

import org.smartfrog.services.anubis.locator.AnubisLocator;
import org.smartfrog.sfcore.common.SmartFrogException;
import org.smartfrog.sfcore.prim.Prim;
import org.smartfrog.sfcore.prim.PrimImpl;
import org.smartfrog.sfcore.prim.TerminationRecord;

public class GroupMonitor extends PrimImpl implements Prim {

    private   AnubisLocator  locator;
    protected String         groupName;
    protected GroupListener  listener;

    public GroupMonitor() throws Exception {
        super();
    }

    public void sfDeploy() throws SmartFrogException, RemoteException  {
        try {
            super.sfDeploy();
            groupName = sfResolve("groupName").toString();
            locator   = (AnubisLocator)sfResolve("locator");
            listener  = new GroupListener(groupName);
            locator.registerListener(listener);
        }
        catch (Exception ex) {
            ex.printStackTrace();
            throw (SmartFrogException)SmartFrogException.forward(ex);
        }
    }
```

```
    public void sfStart() throws SmartFrogException, RemoteException  {
        try {
            super.sfStart();
        }
        catch (Exception ex) {
            ex.printStackTrace();
            throw (SmartFrogException)SmartFrogException.forward(ex);
        }
    }

    public void sfTerminateWith(TerminationRecord tr) {
        locator.deregisterListener(listener);
        super.sfTerminateWith(tr);
    }
}
```

Appendix E StabilityNotices.java

```
/** (C) Copyright 1998-2005 Hewlett-Packard Development Company, LP

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For more information: www.smartfrog.org

*/
package org.smartfrog.services.anubis.components.examples;


import java.rmi.RemoteException;

import org.smartfrog.services.anubis.locator.AnubisLocator;
import org.smartfrog.services.anubis.locator.AnubisStability;
import org.smartfrog.sfcore.common.SmartFrogException;
import org.smartfrog.sfcore.prim.Prim;
import org.smartfrog.sfcore.prim.PrimImpl;
import org.smartfrog.sfcore.prim.TerminationRecord;
import org.smartfrog.sfcore.reference.Reference;


public class StabilityNotices
        extends PrimImpl
        implements Prim {

    public class Stability extends AnubisStability {
        public void stability(boolean isStable, long timeRef) {
            if( isStable ) {
                System.out.println("******  Partition  has  stablized  with
time reference " + timeRef );
            } else {
                System.out.println("****** Partition is UNSTABLE");
            }
        }
    }

    static private Reference LOCATOR_REF = new Reference("locator");
           private AnubisLocator locator = null;
           private AnubisStability stability = null;

    public StabilityNotices() throws Exception { super(); }
```

```
    public void sfDeploy() throws SmartFrogException, RemoteException  {
        try {
            super.sfDeploy();
            locator = (AnubisLocator)sfResolve(LOCATOR_REF);
            stability = new Stability();
            locator.registerStability(stability);
        }
        catch (Exception ex) {
            ex.printStackTrace();
            throw (SmartFrogException)SmartFrogException.forward(ex);
        }
    }

    public void sfStart() throws SmartFrogException, RemoteException  {
        try {
            super.sfStart();
        }
        catch (Exception ex) {
            throw (SmartFrogException)SmartFrogException.forward(ex);
        }
    }

    public void sfTerminateWith(TerminationRecord tr) {
        locator.deregisterStability(stability);
        super.sfTerminateWith(tr);
    }
}
```