The Anubis Service

Paul Murray
Internet Systems and Storage Laboratory
HP Laboratories Bristol
HPL-2005-72
June 8, 2005*

timed model, state monitoring, failure detection, network partition

Anubis is a fully distributed state monitoring and failure detection service designed to report distributed system states in a timely and consistent manner. It is based on a temporal model of distributed computing that includes the notion of network partitions. Here we describe the properties of the Anubis service and the protocols that implement them, along with a description of the Anubis service user interface.

* Internal Accession Date Only
Approved for External Publication

The Anubis Service

Paul Murray
Hewlett-Packard Laboratories, Bristol, Filton Road, Bristol BS34 8QZ, UK
pmurray@hp.com

Abstract

Anubis is a fully distributed state monitoring and failure detection service designed to report distributed system states in a timely and consistent manner. It is based on a temporal model of distributed computing that includes the notion of network partitions. Here we describe the properties of the Anubis service and the protocols that implement them, along with a description of the Anubis service user interface.

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][10][17] 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 adaptation.

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. In either case the ability to interpret the actual state of the system is critical to providing effective dynamic management behaviour.

Observation of actual system state in a dynamic distributed environment is the key point addressed by the Anubis service. Anubis provides the ability to discover objects, monitor their states, and detect their failure or absence from anywhere in the system. These three capabilities are provided under a single state based interface that constitutes our view of actual system state.

State observation allows multiple management agents to take local actions based on local decision processes, but this is only useful if they can exhibit globally consistent behaviour. Some particular points that affect consistency in this context are as follows:

- The management agents may not be aware of any causality relationships among system components. The interaction among components is not tracked by the management agents, so causal ordering cannot be used as a concurrency model.
- Systems may partition, implying an inability for agents to observe components they are separated from. We would like some form of state consistency criteria that respects partitions, which in turn requires the need to monitor communication and track partitioning behaviour.
- Any basic state will have a single source but possibly multiple observers (e.g. management agents). We need to maintain consistency between the component and each observer, and among the multiple observers.
- A management agent may choose to compare state observations to determine satisfaction of a distributed predicate or to infer a distributed state (these two actually being equivalent). So we

need to maintain consistency between multiple distributed state observations such that multiple agents can consistently evaluate distributed predicates.

Anubis implements a temporal consistency model and provides time bound guarantees for disseminating state information across a distributed system. Its state delivery semantics provide management agents with a programming model that allows them to coordinate actions without any interaction other than simple state observation. Additionally, state observation encompasses discovery and failure detection, bringing several management functions under a single framework.

The focus of this paper is the Anubis service itself; we do not cover examples of its use, although it has been used in several adaptive management systems (e.g. [2] and [9]). A discussion of use cases from these systems can be found in [13].

In the next section we describe work that is related to our own, including the foundations that we have built upon and alternative systems that use different approaches. Section 2 describes the approach taken in the Anubis service. In section 4 we provide a formal statement of the Anubis state observation properties and the protocols used to implement them. This section focuses on the communication layers of our service as they provide the base semantics. The service architecture and implementation are outlined in section 5, including the higher level state dissemination arrangement. The programmatic interface is described in some detail in section 6 along with a toy example of its use. Finally, we conclude in section 7.

2 Related Work

Anubis draws most influence from the fail-aware timed model. The timed model [4] characterizes the degree of asynchrony in a distributed system in terms of communication delay, scheduling delay, and rate of clock divergence, and argues that usually a distributed system will operate within a set of time bounds for these parameters and that when it does it exhibits quantifiable timeliness properties. Fail-awareness [5] adds the notion of an exception indicator that identifies when these properties cannot be relied upon, allowing the development of protocols that exploit the timed model. In [6] and [12] the authors describe group membership protocols and (in the later) group communication, based on the timed model.

Anubis is based on a variation of the timed model that uses approximately synchronised system clocks. We define the notion of a partition view, a single node's view of timeliness in communication paths, and define consistency properties that relate the partition views of multiple nodes to one another. Our version of the fail-aware indicator is a stability predicate that links the consistency properties of stable and unstable partitions. This approach provides the base set of semantics for communication within an Anubis system. We then build state dissemination protocols that exploit the base semantics to derive our state consistency properties.

In [11] the authors describe an approach to global predicate evaluation that uses a partial temporal order among states constructed from timestamps obtained from approximately synchronised clocks (known as *rough real time* in their terminology). Their approach allows multiple observers to consistently evaluate predicates over distributed states, providing the ability for multiple management agents to make consistent decisions based entirely on their own observations of distributed states. The state consistency properties of Anubis are intended to support this type of global predicate evaluation.

The Anubis partition view is a type of group membership that has temporal consistency properties. This can be contrasted with the view synchrony properties used by many group membership protocols such as ISIS [3] or Totem [1]. View synchrony defines an order constraint relative to group membership changes that is established using commit-style protocols among group members. Anubis partition properties are based on a partial order obtained by monitoring the timeliness of communication paths in the background. Anubis has a constant communication overhead in both fail-free and failure cases.

The state dissemination arrangement in Anubis might be compared to publish-subscribe event notification systems such as TIB/Rendezvous [14]. However, Anubis does not perform event notification, it performs state observation. The important difference is that Anubis provides an interpretation of the current state of objects in the system, not their past transitions, and has semantics based on this notion.

Astrolabe is a scalable distributed system monitoring service described in [16]. Astrolabe directly addresses scalability in a way that our service does not. It is modelled on DNS, collecting information in zones with the additional ability to propagate aggregated information between zones. Its programming model is based on standard database interfaces and an extended form of SQL to define aggregation queries that determine how information is propagated.

Astrolabe has a consistency model called *eventual consistency*, a weaker model then our temporal consistency. Astrolabe does not identify when eventual consistency has been achieved and it is not achieved in the event of network partitions. However, the weakness of this model is deliberate as it supports their objective of high scalability.

High availability cluster management systems such as HP MC/ServiceGuard [8] include functionality comparable to that of Anubis. Most of these systems use a form of replicated database to store system configuration (including application configuration) using a transactional consistency model. They are very much targeted towards failover recovery management and do not scale beyond a few tens of nodes. The functionality they provide can be implemented by management agents that use Anubis to observe system state. Although this approach would result in similar functionality it targets larger and more flexible environments.

3 Anubis

We developed the Anubis service to provide a simple state-based interface suited to automated adaptive management agents. In our approach each part of the computing system represents its own state; there is no separate copy maintained in a repository, database, or centralized server. 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. The service can be viewed as part of an application management framework and any component can use the service by implementing the appropriate interface to make its own states accessible and to access the states of other components.

Anubis deals with communication network partitions. When a network partition occurs management agents 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 have constructed a consistency model that deals with asymmetric partitions as well as partitions with symmetric communication paths. To do this we used an approach based on timed communication paths that exploits the timed model as outlined in section 2.

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 one representing a matching provider.

This arrangement is depicted in Figure 1 below. A component that wants to make part of its state visible through Anubis it does so by encapsulating that state in a provider object (as done by component Y). If a component wants to observe a state in the system it does so via a listener object (as done by component X).

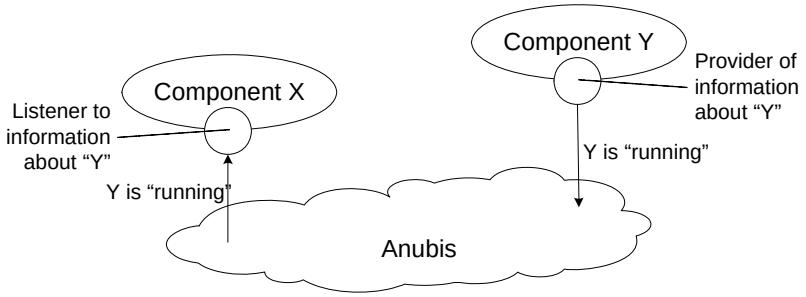

Figure 1 Component interaction with Anubis

Throughout this paper we use the terms provider, listener, state, and state observation. We emphasise that providers have a state and listeners observe states. We will define temporal consistency proprieties relating state observations.

3.2 Partitions and Stability

The consistency properties provided by Anubis are based on a notion of timeliness in communication paths. The Anubis service is implemented by a peer-group of servers spread across a distributed system. Each node monitors its communication paths with its peers and determines a timeliness relation that we call its *partition*. The nature of a partition is that it exhibits synchronous behaviour (by definition – it is defined as a communication relation that is timely).

Under normal circumstances all peers will behave in a timely manner and so will all observe the same partition. When this appears to be the case, a server will determine that its partition is *stable*. A stable partition is not only synchronous, but transitively so.

Anubis disseminates state information within partitions. The nature of partitions and stability means that one peer server can determine how long it takes to perform the dissemination and so establish a temporal consistency property for state observation.

At times the peer-group may be unstable or go through membership transitions due to new peers turning up or leaving, or due to network partitions, or failures. At these points the consistency properties would be disrupted. Moreover, even when a peer determines that the system is stable, it can only really be sure that it *has been* stable so far, not that it will remain stable. The partition properties and the resultant state observation consistency properties we describe deal with these cases.

The state observation properties we describe support the following capabilities:

- **Time-bound Observation** – a listener will discover the current state of a provider within a known period of time.
- **Non-existence Observation** – if a provider leaves the partition (due to failure or network problems) a listener will observe its absence within a known period of time. More generally, when stable, a listener can determine that a matching provider does not exist within a known period of time.
- **Distributed Predicate Evaluation** – the states observed by listeners can be correlated with respect to time.
- **Group Membership** – a listener observes the state of all providers that use the same name. So the listener implicitly knows the collection of providers, and hence Anubis implements group membership.
- **Mutually Concurrent Agreement** – two different listeners listening to the same providers will detect consistent 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 observations, without communicating with each other.

In the next section we formally define the Anubis observation consistency properties and the protocols that implement them.

4 Model

We model a distributed software system that runs on a computer infrastructure of compute nodes connected by communication networks. The model is based on the notion of a processes that run on compute nodes and time values that are provided by the compute nodes (system clocks). The model assumes processes can communicate via the networks using a message passing paradigm, but attempts to abstract that communication by referring to properties of the information passed in this way. The model is indexed by a notional real time that is not explicitly observable by any component being modelled. The basics of the model are as follows.

A process is a finite processing agent. We use Λ to denote the set of all possible processes and *System* to denote a finite set of processes that is fixed to represent a given system. A process has a sequence of current states indexed by a notional real time *RT* modelled using the set of natural numbers $\mathbb{N}$: $p \in \Lambda = \langle \sigma_p(t), \sigma_p(t+1), \ldots, \sigma_p(t+n) \rangle$ where $\sigma_p(t)$ is the current state of process *p* at time *t*.

For simplicity we assume that real time has sufficient granularity to capture each subsequent state of a process in a different time slot, but a single state may persist for multiple time slots. Communication between processes is represented by changes in state that encapsulate the information that has been communicated. We preserve causality by insisting that information cannot be obtained by the receiving process before it exists at the sending process.

Each process has as part of its state a local clock *C* that provides local time values. These values are monotonically increasing with real time and are again modelled using the set of natural numbers. We make a fundamental assumption about local clocks: they have a bounded drift rate ρ from real time that is small in the sense that it is insignificant over short time intervals. Hence, over short intervals we approximate the progress of a local clock to the progress of real time:

$$C_p(t) + \alpha = C_p(t+\alpha) + \alpha\rho \simeq C_p(t+\alpha) \text{ where } \alpha \text{ is small}$$

Naturally, "small" is open to interpretation. In the Anubis context, critical time measurements relate to intervals of no more than a few seconds, or at most minutes. We propose that the clock drift over these time intervals is negligible; as supported by the analysis in [4]. In practice clock drift will be controlled using a network time protocol.

In the following we will use the symbols *p*, *q* and *z* to represent processes, *r*, *s*, and *t* to represent real time values, and δ, μ and ω to represent quantities of real time (real time values are additive). We also use the notation (r,s) to represent open intervals of real time, $[r,s]$ to represent closed intervals, and mix the two (e.g. $[r,s)$ is closed at the start and open at the end).

4.1 Observation Consistency

The Anubis service gives the user the ability to define state providers that are open for observation, and state listeners that can observe the state associated with providers anywhere in the system. Providers and listeners reside at processes and are encoded in their state. A listener observes a provider so long as the state information it represents can be transferred to the listening process under a given consistency constraint. We define $observable_p(t)$ to be the set of providers that can be observed by a process *p* at real time *t* with the following consistency constraint.

(1) $\forall p,q \in System, \forall t \in RT \bullet$

$stable_p(t) \Rightarrow$

$\left(observable_q(t) \subseteq observable_p(t)\right) \vee \left(observable_q(t) \cap observable_p(t) = \varnothing\right)$

The constraint uses a stability predicate $stable_p(t)$ that is defined later. Stability is a property of communication relative to the process *p* at time *t*.

This constraint means that if a process is stable, then each other process in the system can observe a subset of the providers that it can, or a disjoint set of providers. They cannot observe a partially overlapping set of providers or a superset.

4.2 Heartbeat Protocol

The heartbeat protocol determines timeliness of one way connections from other processes. The notions of a timely process and a timely connection from that process are synonymous. Timeliness is defined as follows:

(2) $Let: \mu =$ a fixed length of time

(3) $Let: \omega > 2\mu$

(4) $\forall p,q \in System, \forall t \in RT \bullet$

$recentHeartbeat_p(q,t) \Leftrightarrow$

σ_p(t) contains state representing a heartbeat from q timestamped $\geq C_p(t) - \mu$

(5) $\forall p,q \in System, \forall t \in RT \bullet$

$quiescing_p(q,t) \Leftrightarrow$

$\exists r,s \in RT \bullet \left(t - \omega \leq r < s < t\right) \wedge \left(timely_p(q,r) \wedge \neg timely_p(q,s)\right)$

(6) $\forall p,q \in System, \forall t \in RT \bullet$

$timely_p(q,t) \Leftrightarrow \left(recentHeartbeat_p(q,t) \wedge \neg quiescing_p(q,t)\right) \vee \left(p = q\right)$

Processes always have their own state by definition, so they are always timely relative to themselves (as indicated by the last term in (6) above). Note that we specify everything in terms of real time. Heartbeats are time stamped using local clock times, and the definition of *recentHeartbeat* uses a comparison of time stamps to local time, however it defines timeliness in terms of real time. Definition (4) is left informal deliberately as we believe its meaning is clear.

In addition to timestamps, the heartbeat messages can piggy-back other state information. Upper protocol layers maintain state information that is communicated in heartbeats. Each heartbeat carries only the version of the state information that was current when it was sent. We use *I* to denote this information. The information determined by process *p* about process *q* at time *t* is denoted $I_p(q,t)$. State information passed in heartbeats has the property of being timely:

(7) $\forall p,q \in System, \forall t \in RT \bullet$

$\neg timely_p(q,t) \Rightarrow I_p(q,t) = \perp$

(8) $\forall p,q \in System, \forall t \in RT \bullet$

$$timely_p(q,t) \Rightarrow \exists s \bullet s \le t \bullet I_p(q,t) = I_q(q,s)$$

Note the subtlety here: not all state information will be communicated, even if it is timely. Only information that was current at the time a heartbeat is sent gets communicated. In addition, unreliable transports may omit heartbeats. We assume reliable transports do not omit heartbeats and deliver them in order when they are timely. We assume reliable transports provide no guarantees at all when untimely.

4.3 Connection Set

We define the connection set and connection set counter as follows:

(9) $\forall p,q \in System, \forall t \in RT \bullet$

$$CS_p(p,t) = \{q \mid timely_p(q,t)\}$$

(10) $\forall p,q \in System, \forall s,t \in RT \bullet$

$$s < t \wedge CS_p(p,s) \neq CS_p(p,t) \Rightarrow CSCounter_p(p,s) < CSCounter_p(p,t)$$

(11) $\forall p,q \in System, \forall t \in RT \bullet$

$$CS_p(p,t) = CS_p(p,t+1) \Rightarrow CSCounter_p(p,t) = CSCounter_p(p,t+1)$$

By substituting *CS* and *CSCounter* for state information *I* in (7) and (8) in the heartbeat protocol we can see that each process has access to connection sets and connection set counters for each process it considers timely: i.e. the ones in its own connection set.

The connection set counter allows one process to detect that another process' connection sets have changed, even if the change is not reflected in the connection set delivered in the heartbeat. This is how we deal with the fact that not all connection set changes will be communicated.

4.4 Stability

We define stability as follows:

(12) $Let : \delta = 2\mu$ (the length of the stability interval)

(13) $\forall p \in System, \forall t \in RT \bullet$

$$consistent_p(t) \Leftrightarrow \forall q \in System \bullet q \in CS_p(p,t) \Rightarrow CS_p(q,t) = CS_p(p,t)$$

(14) $\forall p \in System, \forall t \in RT \bullet$

$$stable_p(t) \Rightarrow consistent_p(t)$$

(15) $\forall p \in System, \forall t \in RT \bullet$

$$stable_p(t) \Rightarrow \forall q \in CS_p(p,t), \forall s \in (t-\delta, t] \bullet$$
$$CSCounter_p(q,s) = CSCounter_p(q,t)$$

In other words, the process is stable when all the connection sets it knows about are consistent (14) and have all remained unchanged for the interval δ (15). We call the δ the stability interval. Notice that stability is relative to a process.

4.5 Partition Protocol

Each process has a view of the system that may change over time called its partition. Partitions satisfy the following property (the partition property):

(16) $$\forall p,q \in System, \forall t \in RT \bullet$$
$$stable_p(t) \Rightarrow \left(partition_q(t) \subseteq partition_p(t) \right) \vee \left(partition_q(t) \cap partition_p(t) = \varnothing \right)$$

The partition protocol derives process stability and its partition. The protocol operates by communicating information about timely connections in heartbeats.

The partition is established by maintaining the largest set that satisfies the following rules:

(17) $$\forall p \in System, \forall t \in RT \bullet$$
$$stable_p(t) \Rightarrow partition_p(t) = CS_p(p,t)$$

(18) $$\forall p \in System, \forall t \in RT \bullet$$
$$partition_p(t) \subseteq CS_p(p,t)$$

(19) $$\forall p,q \in System, \forall t \in RT \bullet$$
$$p \notin CS_p(q,t) \Rightarrow q \notin partition_p(t)$$

(20) $$\forall p \in System, \forall s,t \in RT \bullet$$
$$\left(s < t\right) \wedge \left(\forall r \in \left[s,t\right) \bullet \neg stable_p(r) \right) \Rightarrow partition_p(t) \subseteq partition_p(s)$$

Definition (20) states that the partition of a process can only shrink so long as the process is unstable. The only time a partition can grow is when it is stable at which point it becomes equal to the connection set (17). Rules (18) and (19) define further constraints on the partition.

The Appendix contains proof that these rules construct a partition view that satisfies the partition property (16).

4.6 Leader Election

We associate a leader $leader_p(p,t)$ with each process that satisfies the following property:

(21) $$\forall p \in System, \forall t \in RT \bullet$$
$$stable_p(t) \Rightarrow \forall q \in partition_p(t) \bullet stable_q(t) \Rightarrow leader_p(t) = leader_q(t)$$

(22) $$\forall p \in System, \forall t \in RT \bullet$$
$$stable_p(t) \Rightarrow leader_{leader_p(t)}(t - \delta) = leader_p(t)$$

The first of these properties states that all stable members of the same partition agree on the leader. The second states that the leader of a stable process already selected itself leader, regardless of stability. Note that the second does not state that the selected process still considers itself leader.

The simplest leader election protocol that supports these properties is based on a total order among processes: the process with the greatest order in the partition is the chosen leader. The proofs for this protocol are trivial. The first can be derived from the partition property (16) and the second from lemma (24) in the appendix.

4.7 State Dissemination

We can state the following time-bounded state dissemination property for communication in partitions:

(23) $$\forall p, q \in System, \forall t \in RT \bullet$$
$$q \in partition_p(t) \Rightarrow \exists s \in (t-\delta, t) \bullet I_p(q,t) = I_q(q,s)$$

This property states that any information passed within a partition is known to be up no more than δ time units old. The property is proved for our partition protocol in the appendix.

We noted earlier that communication using heartbeats over an unreliable transport does not guarantee that information will not be omitted, only that any information received is timely. However, if we use ordered, reliable communication transports then all information sent in heartbeats will be received in order within the known time bound so long as the communication is timely.

We consider a provider (or rather the provider's state value) to be observable by a process p at time t if the provider resides at a process $q \in partition_p(t)$, and we use $observable_p(t)$ to denote the set of provider state values that are observable by process p at time t. Hence the observation consistency property (1) is derived directly from the partition property (16). We restate the observation consistency property here:

$$\forall p, q \in System, \forall t \in RT \bullet$$
$$stable_p(t) \Rightarrow$$
$$\left(observable_q(t) \subseteq observable_p(t)\right) \vee \left(observable_q(t) \cap observable_p(t) = \varnothing\right)$$

This property is distinct from actual observation which is a property of listeners not processes. Just because a process is able to observe a provider, it does not mean it contains a listener for that provider, or that the associated state value for that provider has been communicated.

State information is passed in heartbeats as required, so state observation is subject to the time-bounded state dissemination property above. In essence, a provider becomes *observable* by a listener as soon as it enters the listeners partition. It will actually be *observed* within δ time units (so long as it remains in the partition).

In practice we communicate provider state values between processes on demand: we need to locate and bind processes that need to disseminate information. This is done using an arrangement that exploits the leader process as described in the implementation section below. This discovery step extends the initial time bound for observation of a provider state value to 3δ. However, this is the upper limit and once done the time-bounded state dissemination property above applies.

5 Implementation

The Anubis service is implemented entirely in Java as a peer-group of servers using a layered architecture shown in Figure 2 below. A server in the implementation directly corresponds to a process in the system model described in the previous section. At the lowest level the UDP and TCP protocols are used to implement multicast and point-to-point ordered communication respectively.

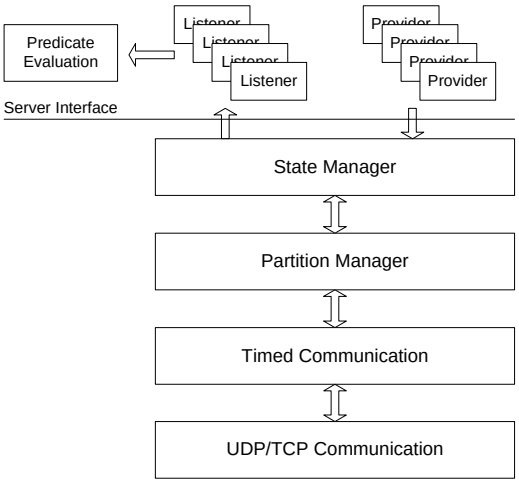

Figure 2 Anubis server architecture.

5.1 Timed Communication

The next layer implements timed communication between servers and provides connection based messaging to the upper layers. Anubis assumes the existence of a separate clock synchronization protocol to keep system clocks aligned. The communication path between servers is considered timely if the servers are able to exchange messages within the chosen time bound δ. Timeliness is checked by using time stamps to measure the transmission delay of heartbeat messages sent at regular intervals (of less than μ). The value of δ is chosen to allow for a reasonable clock skew, transmission delay, and processor scheduling delay, each of which contribute to a perceived communication delay. We say that a server is timely with respect to the local server if its communication is timely. A server is always timely with respect to itself.

When there are no other messages to send, heartbeats are transferred using UDP multicast. A single heartbeat is delivered to all servers listening on the multicast group and used for measuring the delay from its source. The protocol creates a messaging overhead that is linear with the number of servers and dependent on the frequency of heartbeats.

When upper layers wish to send messages to other servers they request that a message connection is established. At this point the timed communication layer switches to using a TCP connection with that particular server. This connection is also timed and provides reliable, ordered delivery. Message connections are removed when they are no longer needed as they add to the communication overhead.

The timed communication layer presents a view of all the servers that are currently timely; this is the connection set defined in section 4.3. The timed communication layer will only open message connections with servers in its connection set. All errors on a message connection are interpreted as untimely communication – inability to deliver messages on time. If a message connection becomes untimely it drops out of the connection set. So the timed communication layer provides a very simple use model for messaging: either a server is in the connection set and any messages sent to it (or received from it) are delivered in a known time bound δ or it is dropped from the connection set. All communication errors are represented by a change in the connection set. No other error notifications are given.

Lastly, the timed communication layer piggy-backs a copy of its current connection set with an associated time stamp each time it sends a heartbeat message. This information is used by the partition manager.

5.2 Partition Manager

The partition manager layer implements a protocol that derives the partition view with the properties defined in section 4.5. Because the timed communication layer piggy-backs its connection set on heartbeats, the partition manager has access to a "recent" connection set from each of the timely servers, where "recent" is defined as no older than δ time units. The partition manager uses these connection sets to derive the partition view, and its stability, according to the rules defined in section 4.5, as follows:

1. If there is a change in the connection sets and they are all consistent (i.e. the same), stamp the local connection set with the time $now+\delta$, mark the partition view unstable, but do not change the membership.
2. If the connection sets are consistent and unchanged and the local clock time has reached the recorded time stamp, set the partition view equal to the local connection set and mark it stable.
3. If a remote server drops out of the local connection set or the local server drops out of a remote server's connection set, remove the remote server from the partition view and mark it unstable.
4. If a new server appears in the connection set, mark the partition view unstable but do not change the membership.
5. If any other changes occur in the connection sets, mark the partition view unstable but do not change the membership.

1 identifies that the connection set may be transitive, but due to the communication delay we cannot be certain that all servers have recognized this fact yet; some may even think the partition view is larger. 2 defines stability: at this point all servers in the partition view definitely have observed the transitivity and definitely have partition views that are subsets of the local connection set.

1, 3, 4, and 5 all demonstrate that any change in the connection sets represents instability. We note that there is only one condition under which a partition view may get larger, and that is when it becomes stable in 2. This is where the partition view differs from the connection set reported by the timed communication layer, and enforces the property that if one server is stable with partition view P then the partition views of all other servers are either subsets of P or do not intersect P.

The partition manager is also responsible for the leader election protocol. The rules defined in section 4.6 are used to select a candidate that is in the local partition view. All servers with the same partition view elect the same leader regardless of stability.

The partition view protocol does not change the communication pattern of the timed communication layer at all. All its communication is provided by data piggy-backed on regular heartbeat messages, so there is no messaging overhead for the protocol and minimal processing overhead. By encoding information such as connection sets as bit sets, we are able to cover more than a thousand Anubis servers in a single MTU payload.

Lastly, the partition manager layer also generates a partition base time reference. All connection sets are time stamped. When the local server is unstable its base time reference is undefined, but when stable the reference is equal to the latest connection set time stamp. All stable servers in the same partition will agree on the base time reference. This can be used as a basis for consistency when correlating among users when correlating similar distributed state predicates.

5.3 State Manager

The state manager layer sits on top of the partition manager layer and provides state dissemination between providers and listeners within partitions. The state manager implements a very simplistic location protocol. The server that is elected leader by the partition manager becomes a global locator

and all servers register the sets of provider names and listener names they contain. The global locator informs servers with providers which servers hold corresponding listeners. A server with a provider is responsible for forwarding the providers state value to any matching listeners. If a provider (or listener) deregisters its server informs matching listeners (or providers).

All these message exchanges are one way. The fact that the communication paths are timed implies reliable delivery within the known time bound. If there is a change in the partition membership the state manager will treat remote providers and listeners on servers now outside the local partition as if they had deregistered.

The delivery of state information among providers and listeners is based on the partition view reported by the partition manager and implements the state consistency properties defined.

Our implementation adds time stamps to state values when they change. The approximately synchronized clocks allow comparison of independent states as described in [11]. New states are transferred immediately from the provider to all matching listeners and arrive within the known time bound δ. This is sufficient to perform distributed predicate evaluation over observed states with the same consistency properties of state values. The partition base time reference defines a time at which all servers have a known consistency, providing a simple means for all servers in a single partition to perform consistent evaluations.

6 Programmatic Interface

There is a single Java interface class for the Anubis service called AnubisLocator. In addition there are three Java base classes that represent providers, listeners and values called AnubisProvider, AnubisListener and AnubisValue respectively, and an interface called AnubisStability.

6.1 AnubisLocator

The AnubisLocator interface has six methods that are of interest to the user. These are: registerProvider(...), deregisterProvider(...), registerListener(...),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 and is described in the next section. When a provider has been registered with the Anubis service using registerProvider(...) it becomes visible to the service and observable within its partition, so 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, described in section 6.3. 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 behaviour becomes undefined).

```
void registerListener(AnubisListener listener);
void deregisterListener(AnubisListener listener);
```

The last two methods register and deregister a stability notification interface. The signatures are given below and the interface is described in section ???. This interface is used to inform the user when the server becomes stable or unstable, and the partitions base time reference.

```
void registerStability(AnubisStability stability);
void deregisterStability(AnubisStability stability);
```

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.

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);
```

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 by passing their serialized form in messages.

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 on the observable set of states in the local partition. 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. These changes will occur according to the time-bounded state dissemination property (taking the initial discovery delay into account). A new AnubisValue will appear within 3δ of there being an associated provider in the partition so long as the local Anubis server is stable. Updates of values already in the collection will occur within δ of the associated provider being updated regardless of stability. Values will be removed from the collection within δ of the associated provider exiting the local server's partition, whether due to deregistering or failure and regardless of stability.

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 observable. This occurs as follows:

1. For each provider that is observable 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 and it will be created within 3δ of the provider's appearance in the local partition, so long as the local server is stable.
2. 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. This will happen within δ of the change occurring, regardless of stability.
3. 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. This will happen within δ of the provider's disappearance.

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. 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 included in the AnubisListener class for 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 time-bounded state dissemination properties, 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

AnubisStability is an interface used to notify users when the local server becomes stabile or unstable and the base time reference for the partition when stable. The interface contains a single method:

```
void stability(boolean stable, long timeRef);
```

When the interface is registered the local server uses it to make an initial call that indicates its current stability status and then makes subsequent calls when its status changes. If the server is stable the timeRef parameter will be the base time reference for the local partition. If unstable the timeRef parameter will be set to -1.

6.6 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 an object that observes a collection of objects in the system and their status. In this case the status will indicate if they are in the “initialising” or “started” states in their 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 objects that are going to be observed use the provider interface. The object has three simple steps to implement:

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

   ```
   locator = (AnubisLocator)sfResolve("anubisLocator");
   ```

   We assume here that the object has access to a configuration/service interface location system that finds the Anubis server interface. In our example we are using SmartFrog [2][15].

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

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

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

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

These steps are performed during an initialisation phase and result in the object state represented by the value “initialising” being visible in the Anubis service under the name held in the by the variable myName.

During a start phase the value of the provider is set to “started” to indicate its new state. When terminating, the object deregisters the provider from the Anubis service.

The group membership monitor specialises the AnubisListener base class and uses a specialised version of the AnubisValue class. These are implemented as the GroupListener and GroupMember classes below.

```
Public GroupListener extends AnubisListener {

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

   public void newValue(AnubisValue v) {
     System.out.println("New value in group " + getName());
     ((GroupMember)v).newValue();
   }

   public void removeValue(AnubisValue v) {
     System.out.println("Removed value from group " + getName());
     ((GroupMember)v).removeValue();
   }

}
```

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 shown.

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.

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 GroupMember extends AnubisValue {

   private GroupListener group;

   public void GroupMember(providerInstance i, GroupListener group) {
     super(i);
     this.group = group;
   }

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

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

}
```

Any object that wants to observe a group of objects using the group listener class simply constructs an instance of GroupListener and registers it with Anubis as follows:

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

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 in the group. For example, if listening to the name "fred" and an object initialises itself creating the provider as shown above the listener would output:

```
New value in group fred                    (from GroupListener)
New value in group fred = initialising     (from GroupMember
                                                – in initialising  state)
```

And the object is started the listener would output:

```
New value in group fred                    (from GroupListener)
New value in group fred = started          (from GroupMember
                                                – in started state)
```

Notice 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 object that was initialising that is now started. Note also that the values that have been passed are the strings "initialising" and "started". Any serializable objects can be used as values; they do not need to be strings. If the user wants to distinguish the identity or location of the objects being observed, this information should be encoded in the state values. Likewise, the user could obtain a remote interface to the object being observed if a reference to it is encoded in the state value.

7 Conclusion

We have described the implementation of the Anubis state monitoring service and the semantics that it provides. Anubis enables multiple user agents spread across a distributed system to attain a consistent view of the state of objects in the system according to a temporal consistency model, providing a sufficient foundation for fully distributed management of distributed systems.

Acknowledgements

The implementation of the partition protocol that provides the partition view of the Anubis service was largely influenced by a prior prototype implemented by Roger Fleming of HP.

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, *Managing MC/ServiceGuard*, http://docs.hp.com/hpux/ha/, part B3936-90073, June 2003

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

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

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

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

[13] P. Murray, "A Distributed State Monitoring Service for Adaptive Application Management", *The International Conference on Dependable Systems and Networks (DSN-05)*, June 2005

[14] 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

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

[16] R. Van Renesse, K. Birman, W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining", *ACM Transactions on Computer Systems*, vol. 21(2), May 2003, pp.164-206

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

Appendix – Partition Protocol Proofs

We wish to prove that the protocol defined by rules (17) to (20) in section 4.5 result in a partition that satisfies the property defined in (16). We start with the relationship between a process's partition and the evolution of its connection set and then move onto the relationship between different process' partitions and their connection sets.

Lemma: information from a process in the partition is no older than δ. This is the time-bounded state dissemination property (23) restated here:

$$\forall p,q \in System, \forall t \in RT \bullet$$
$$q \in partition_p(t) \Rightarrow \exists s \in (t-\delta, t) \bullet I_p(q,t) = I_q(q,s)$$

Proof: This proof relies on our assumption that clocks have negligible drift from real time over short intervals. Assume $q \in partition_p(t)$. Then partition rule (19) implies $p \in CS_p(q,t)$. Substituting connection sets *CS* for the information *I* in (8) we get:

$$\forall p,q \in System, \forall t \in RT \bullet$$
$$timely_p(q,t) \Rightarrow \exists s \bullet s \le t \bullet CS_p(q,t) = CS_q(q,s)$$

And so $p \in CS_p(q,s)$. Choose $s = t - \alpha$ where $\alpha \ge 0$ and let $C_q(s)$ be the timestamp of the heartbeat that carried the information. By the definition of timeliness (6) and recent heartbeat (4) we get:

$$C_q(t-\alpha) \ge C_p(t) - \mu$$
$$\Rightarrow C_q(t) - \alpha + \mu \ge C_p(t)$$

We also know $timely_q(p, t-\alpha)$ and so by the same logic we can choose an earlier time $t - \alpha - \beta$ where $\beta \ge 0$. This time the timeliness definition gives:

$$C_p(t-\alpha-\beta) \ge C_q(t-\alpha) - \mu$$
$$\Rightarrow C_p(t) - \alpha - \beta \ge C_q(t) - \alpha - \mu$$
$$\Rightarrow C_p(t) \ge C_q(t) + \beta - \mu$$

By substitution these two inequalities give:

$$C_q(t) - \alpha + \mu \ge C_q(t) + \beta - \mu$$
$$\Rightarrow \alpha + \beta \le 2\mu$$

So, we have $0 \le \alpha \le 2\mu = \delta$ giving a lower bound on the value of *s*. This lower bound generalises to all information transferred in the same heartbeat, completing the proof.

Lemma: if the process is stable then its connection set has remained unchanged for δ time units and is the same as the partition:

(24) $$\forall p \in System, \forall t \in RT \bullet$$
$$stable_p(t) \Rightarrow \forall s \in (t-\delta, t) \bullet partition_p(t) = CS_p(p,s)$$

Proof: by the definition of connection counters (10) and (11), and stability in (15) we get:

$$CS_p(p,t) = CS_p(p,s)$$

Then rule (17) completes the proof.

Lemma: at all times a process's partition is a subset of any version of its connection set that occurred in the past δ time units:

(25) $$\forall p \in System, \forall t \in RT \bullet$$
$$\forall s \in (t-\delta, t) \bullet partition_p(t) \subseteq CS_p(p,s)$$

Proof: we cut this proof three ways: in the first case we assume the process is unstable throughout the interval; in the second, the process is stable at time *t*; and in the third, the process is stable at some point, but not at time *t*.

The first case assumes instability throughout the interval, so we use rule (20):

$$\forall p \in System, \forall s,t \in RT \bullet$$
$$(s < t) \wedge (\forall r \in [s,t) \bullet \neg stable_p(r)) \Rightarrow partition_p(t) \subseteq partition_p(s)$$

By restricting *s* to be values between *t*-δ and *t* we get:

$$\forall p \in System, \forall t \in RT \bullet$$
$$\forall s \in (t-\delta, t) \bullet (\forall r \in [s,t) \bullet \neg stable_p(r)) \Rightarrow partition_p(t) \subseteq partition_p(s)$$

Clearly our assumption renders $\forall r \in [s,t) \bullet \neg stable_p(r)$ true, so this reduces to:

$$\forall p \in System, \forall t \in RT \bullet$$
$$\forall s \in (t-\delta, t) \bullet partition_p(t) \subseteq partition_p(s)$$

Rule (18) gives $partition_p(s) \subseteq CS_p(p,s)$ which completes the proof for the first case.

The second case assumes the process is stable at time *t*. This case is proved by lemma (24).

The third case assumes the process is stable at some time during the interval, but not at time *t*. By choosing *r* to be the last such time we divide the interval into two: $t-\delta \le u \le r$ and $r < v \le t$. The first interval is proved by the second case above and the second interval is proved by the first case above. This completes the proof for the lemma.

Lemma: if a process is in a stable partition, its connection set was equal to the partition at a time exactly δ in the past.

(26) $$\forall p,q \in System, \forall t \in RT \bullet$$
$$q \in partition_p(t) \wedge stable_p(t) \Rightarrow CS_q(q, t-\delta) = partition_p(t)$$

Proof: from (10) relating connection sets to connection set counters and (14) and (15) regarding stability, we can see that the connection sets known to *p* are all equal and the connection set counters have remained unchanged at least since *t*-δ. For any $q \in partition_p(t)$ we get:

$$CS_p(q, t-\delta) = CS_p(q,t)$$
$$CSCounter_p(q, t-\delta) = CSCounter_p(t)$$

Considering the information known to *p* at time *t-δ* we can see that timeliness property (8) implies:

$$\exists s \bullet s \le t-\delta \bullet CS_p(q,t-\delta) = CS_q(q,s)$$

By similar logic we get $CSCounter_p(q,t-\delta) = CSCounter_q(q,s)$ for the same value s. Considering the information known to *p* at time *t* we can see that lemma (23) implies:

$$\exists s' \bullet t-\delta \le s' \le t \bullet CS_p(q,t) = CS_q(q,s')$$

And by similar logic we get $CSCounter_p(q,t) = CSCounter_q(q,s')$ for the same value s'. Clearly the connection set at *q* remained unchanged throughout $s' \le t-\delta \le s$. From rule (18) we get $partition_p(t) = CS_p(p,t)$ completing the proof.

Corollary: if two processes are stable at the same time they either have identical partitions or disjoint partitions.

(27) $$\forall p,q \in System, \forall t \in RT \bullet$$
$$stable_p(t) \wedge stable_q(t) \Rightarrow$$
$$\left(partition_p(t) = partition_q(t)\right) \vee \left(partition_p(t) \cap partition_q(t) = \varnothing\right)$$

Proof: consider a process $x \in \left(partition_p(t) \cap partition_q(t)\right)$. By lemma (26) we get:

$$CS_x(x,t-\delta) = partition_p(t)$$
$$CS_x(x,t-\delta) = partition_q(t)$$

So either $partition_p(t) = partition_q(t)$ or there is no member in the intersection.

Lemma: if one process considers another untimely it will not be in the other's partition *δ* time units later – regardless of stability.

(28) $$\forall p,q \in System, \forall t \in RT \bullet$$
$$p \notin CS_q(q,t-\delta) \Rightarrow q \notin partition_p(t)$$

Proof: Process *q* communicates its connection set in heartbeats. We cut this proof into two parts: the case in which *p* might not receive that communication (i.e. when it does not receive timely heartbeats from *q*) and the case in which it does.

In the first case we consider *p* believes *q* is untimely (and so not a member of its connection set) at some point between *t-δ* and *t*:

$$\exists s \in \left(t-\delta, t\right) \bullet q \notin CS_p(p,s)$$

Lemma (25) gives us $partition_p(t) \subseteq CS_p(p,s)$ thus completing the proof for the first case.

In the second case we consider *p* believes *q* to be timely throughout the interval *t-δ* to *t*. For refutation we assume $q \in partition_p(t)$ and then prove this cannot be so. Lemma (23) applied to connection sets gives:

$$\exists s \in \left(t-\delta, t\right) \bullet CS_p(q,t) = CS_q(q,s)$$

By rule (19) and our assumption we get $p \in CS_p(q,t)$ and so $p \in CS_q(q,s)$. We know $p \notin CS_q(q,t-\delta)$ and so $s \neq t-\delta$. Therefore $s > t-\delta$ and the connection set of q must have changed to include p at some point in the interval $[t-\delta, s)$. We will use r to denote the first time this was so and note $r \in [t-\delta, t)$.

We now proceed by proving that there is a period of time immediately prior to r during which p either considers q untimely or observes a connection set from q that does not include itself.

Assume $quiescing_q(p,t-\delta)$. If $p \in CS_q(q,r)$ then by the definition of quiescing (5) the last time that q could have considered p timely is $r-\omega$. If p did not receive a heartbeat sent during the quiescing interval then the latest the preceding heartbeat could have been sent is $r-\omega$. This would be received at a time $r' \leq r-\omega+\delta$ (using lemma (23) again). Noting that $\omega > \delta$ this means $r' < r$ and there is an interval of at least $\omega-\delta$ immediately prior to r during which p considers q untimely, and so $q \notin partition_p(t)$ (breaking both our assumptions). If p did receive a heartbeat sent during the quiescence interval then there is a period immediately prior to r in which p was not a member of the connection set reported by q and so again $q \notin partition_p(t)$, completing the refutation for this case.

Assume $\neg quiescing_q(p,t-\delta)$. Then either q has never considered p timely, and so p has never received a connection set from q that contains p, or the last time it did consider p timely was more than the quiescing period in the past. If so a similar argument to the case for $quiescing_q(p,t-\delta)$ applies. This completes the refutation of $q \in partition_p(t)$ and the proof of the lemma.

Lemma: if a process is stable all other processes in its partition either have partitions that are subsets of its partition or ones that are disjoint from it. This is the partition property (16) restated here:

$$\forall p,q \in System, \forall t \in RT \bullet$$
$$stable_p(t) \Rightarrow \left(partition_q(t) \subseteq partition_p(t)\right) \vee \left(partition_q(t) \cap partition_p(t) = \varnothing\right)$$

Proof: We prove this in two cases. In the first case assume $q \in partition_p(t)$. Because p is stable lemma (26) gives $CS_q(q,t-\delta) = partition_p(t)$. Therefore by lemma (25) we obtain the first term $partition_q(t) \subseteq partition_p(t)$.

For the second case assume the inverse: $q \notin partition_p(t)$. Now consider any other process that is in the partition: let $z \in partition_p(t)$. Again by lemma (26) we have $CS_z(z,t-\delta) = partition_p(t)$ and so $q \notin CS_z(z,t-\delta)$. Therefore by lemma (28) $z \notin partition_q(t)$ and so $partition_q(t)$ and $partition_p(t)$ are disjoint sets.

This completes the proofs for the partition protocol.