Anubis

Seeing is believing

What's the point?

Global coordination among multiple agents that take local actions based on local decisions

- We consider agents that are programmed to react to conditions detected in a system
- Sometimes agents need to take actions that are consistent or coordinated with other agents
- Sometimes parts of the system fails, leading to impossibility issues with respect to detection
- Sometimes the agents that need to be coordinated cannot see each other or the same parts of the system

Consistent state observation = consistent decisions

Introduction

- The Theory
  - Timed Asynchronous Distributed Systems Model
  - Fail-Awareness
  - Partitions
  - State Observation

- The Implementation
  - Timed Connections
  - Connection Set
  - Partition Manager
  - Timed Messages
  - State Manager
  - Interface
  - Tools

The Theory

Timed-Asynchronous Distributed Systems

- A realistic theoretical model of distributed systems
  - Known as “timed model” for short
  - Actually a characterisation of real distributed systems

- A distributed system is:
  - Processing nodes with clocks
  - Processes on processing nodes with access to local clock
  - Datagram networks between processing nodes

- Assumptions
  - Almost always, clocks are monotonically increasing
  - Almost always, clocks have bound drift rate
  - In general, Processes are scheduled with bound delay
  - But, processes can crash
  - In general, datagrams are communicated with bound delay
  - But, datagrams can be lost

Fail-Aware

- Fail-awareness is the ability to know when a given semantic model applies (or the system adheres to the specification)
  - Exploit semantics when they hold
  - Understand when they don't

- Fail-awareness can be used to exploit the timed model in reality
  - Interpret everything as timing properties

- Fail-aware timed model
  - Clocks are monotonically increasing and have bound drift rate
  - Processes are scheduled within bound delay
  - Datagrams are communicated within a bound delay

- If any of the above does not hold set an exception indicator

Partitions

- A system is composed of processing nodes :
  - System – the set of all processing nodes
  - We adopt the timed model except to say clocks have bound divergence (skew)

- We establish a *timeliness* relationship between processing nodes
  We say p considers q to be timely at a given moment iff
  - p receives messages from q within a bound time (δ)
  - p's clock has a bound difference to q's clock (ε)
  - p meets its scheduling delay bound (η)

- We denote timeliness by:
  - $\text{timely}_p(q, t)$ – p consider q timely at time t

- In practice, we approximate timeliness by a time bound for:
  - The difference between a datagram timestamp and the time it is processed in the receiving process
  - $\Delta = \delta + \varepsilon + \eta$
  - There are other measures for timeliness, this is an implementation choice

Partitions

- A connection view is relative to a processing node: The connection view of p is:
  - The collection all processing nodes that are timely wrt p
  - $CView_p(t)$ – the connection view of p at time t

- Each processing node shares its connection view:
  - Passed in messages
  - When timely, messages are received within Δ
  - $timely_p(q, t)$ implies p knows $CView_q(t')$ for some $t\text{-}\Delta \leq t' \leq t$

- A connection set is relative to a processing node:
  - The connection set is the collection of connection views from all the processing nodes in the local connection view
  - $CSView_p(q,t)$ – the connection view of q as known to p at time t
  - $CSView_p(q,t) = CView_q(t')$ for some $t\text{-}\Delta \leq t' \leq t$
  - $CS_p(t)$ – the connection set of p at time t
  - $CS_p(t) = \{ CSView_p(q, t) \mid timely_p(q,t) \}$

Partitions

- Stability is relative to a processing node
  A processing node is stable iff:
  - All connection views in its connection set are the same, and
  - The connection set has remained unchanged for Δ
  - $\text{Stable}_p(t) \Rightarrow$

$$(\forall q \in \text{CView}_p(t) \bullet \text{CSView}_p(q, t) = \text{CView}_p(t) \quad) \wedge$$
$$(\forall s \in [t\text{-}\Delta, t] \bullet \text{CS}_p(s) = \text{CS}_p(t) \quad)$$

- Each processing node derives a partition view with the following properties:
  - If p is stable then the partitions of all nodes are either subsets of its partition or do not intersect it

$$\forall q \in \text{System} \bullet (\text{Stable}_p(t) \Rightarrow$$
$$\text{partition}_q(t) \subseteq \text{partition}_p(t) \vee \text{partition}_q(t) \cap \text{partition}_p(t) = \emptyset)$$

  (Note that if two nodes are stable their partitions are the same or disjoint)

  - If p has remained stable for Δ then all nodes in its partition stabilised with the same partition

$$(\forall s \in [t\text{-}\Delta, t] \bullet \text{Stable}_p(s)) \Rightarrow$$
$$(\forall q \in \text{partition}_p(t) \bullet \text{Stable}_q(t\text{-}\Delta) \wedge \text{partition}_p(t) = \text{parittion}_q(t\text{-}\Delta))$$

Partitions

Stable partitions define the semantic model for our distributed system

Stability is our fail-aware indicator

State Observation

- State is represented by a state variable and its value:
  - The variable is uniquely named and can be created and removed
  - The value is any arbitrary structure and may change
  - Where clear from the context, we use the term *state* to mean either:
    - A particular value of a particular variable, or
    - The state variable
  - By *state change* we mean a change of value for a given variable

- Processing nodes have state variables
  - The variable names are unique
  - The values represent states

- Processing nodes can observe state variables
  - Processing nodes observe their own states
  - Processing nodes can chose to observe the states of other nodes
  - States from within the observers partition can be observed
  - States from outside the observers partition are not observed
  - When an observer cannot observe a state (variable) we say it observes the absence of the state (variable)

State Observation

- Information about states can be communicated within a partition within time Δ
  - Known from partition semantics
  - There is a consistency between the state and its observers:
    - Observers observe the state within Δ, or
    - Observe its absence within Δ
  - There is a consistency between observers of a state:
    - Observers either observe the same states within Δ of each other, or
    - Observe the absence of the state within Δ

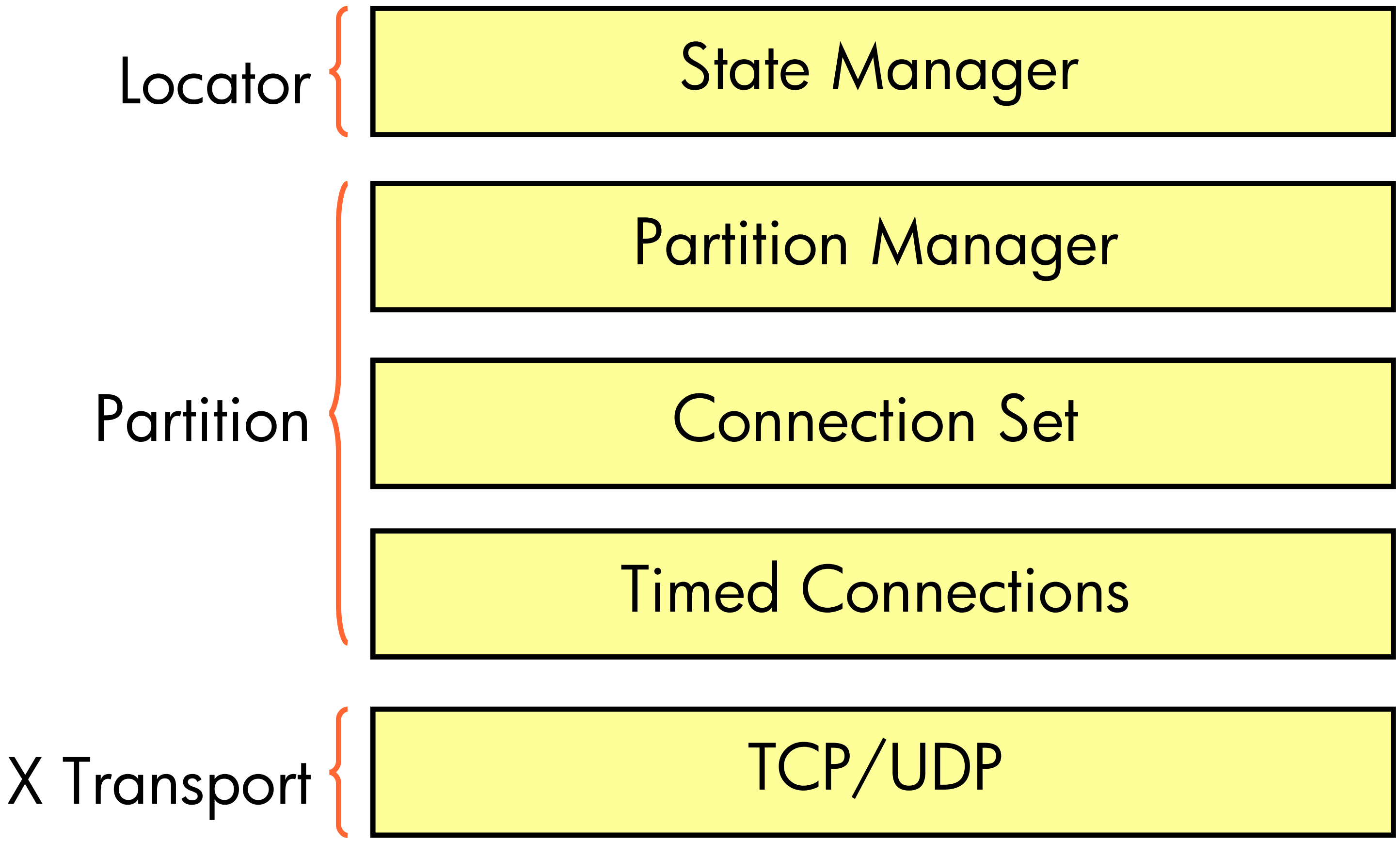
- We say an observer knows about a state that it has observed
  - The knowledge of states that occur Δ or more in the past is consistent among all observers:
    - Either they know the states or not
    - They do not have contradictory knowledge

- Within a stable partition observers can derive something of each others knowledge:
  - When an observer observes a state it does not know if other observers know it
  - After Δ the observer can be sure all observers know it
  - After 2Δ the observer can be sure all observers know all observers know it
  - Etc…
  - When an observer observes a state it knows that any observer not in its partition does not observe the state

- State observation, stability and partitions are sufficient to solve many global coordination problems

The Implementation

Anubis Layers



Locator

State Manager

Partition

Partition Manager

Connection Set

Timed Connections

X Transport

TCP/UDP

View

- A view is a representation of the system relative to a particular viewpoint
  - The viewpoint is always an Anubis server

- A view has:
  - A set of Anubis servers (identified by node id)
  - Stability
  - Timestamp

- Set operations can be performed on views
  - Contains (both node or view), overlap, equal

- There are two important views:
  - Connection view
  - Partition view

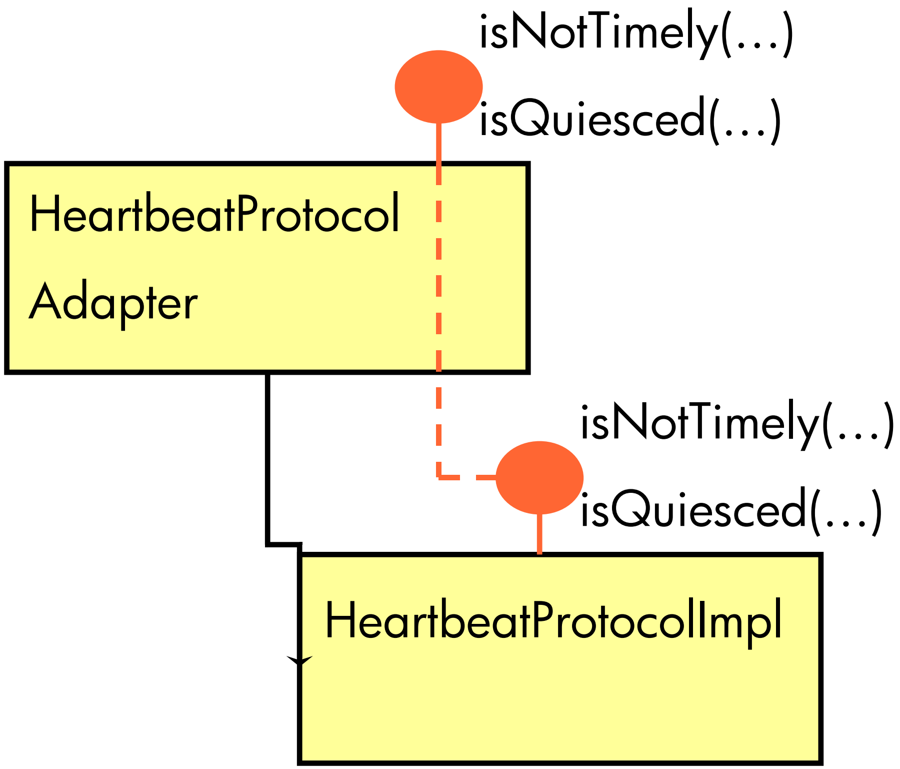
Timed Connections

- A timed connection represents a remote server:
  - Its identity
  - Its timeliness with respect to the local server
  - Its connection view (described later)
  - Its leader candidate

- It is a timing relationship
  - not a message passing relationship
  - But - it can be extended to be a message passing relationship

- A server is always timely with respect to itself
  - It has a timed connection to itself

Timed Connections

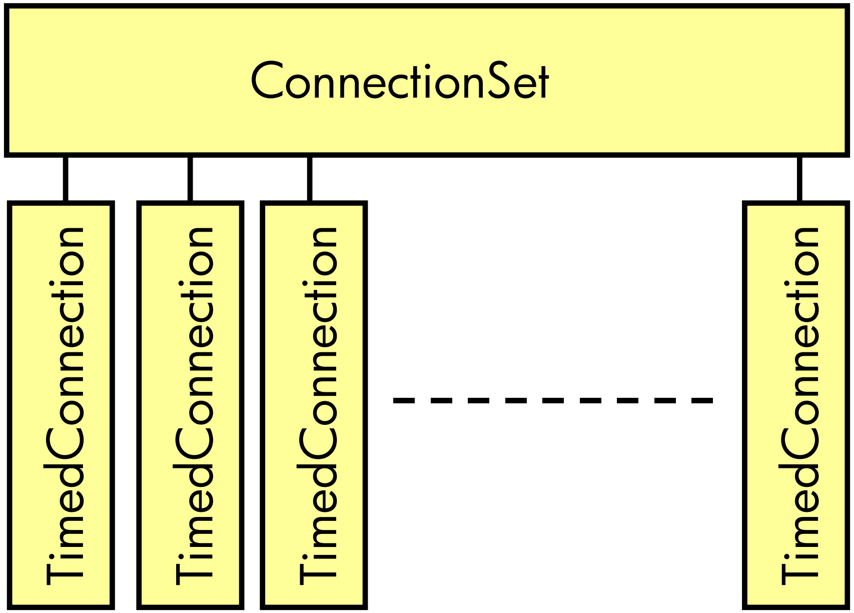


- Connections implement HeartbeatProtocol
  - *HeartbeatProtocol extends View*
- HeartbeatProtocol determines timeliness and quiescence
  - Timely ⇒ meeting timing reqs
  - Quiesced ⇒ gone quiet
- Anubis connections extend HeartbeatProtocolAdapter
  - Protocol is a plug-in
  - Default impl can be replaced
  - Easy to pass protocol state from one connection to another

Connection Set

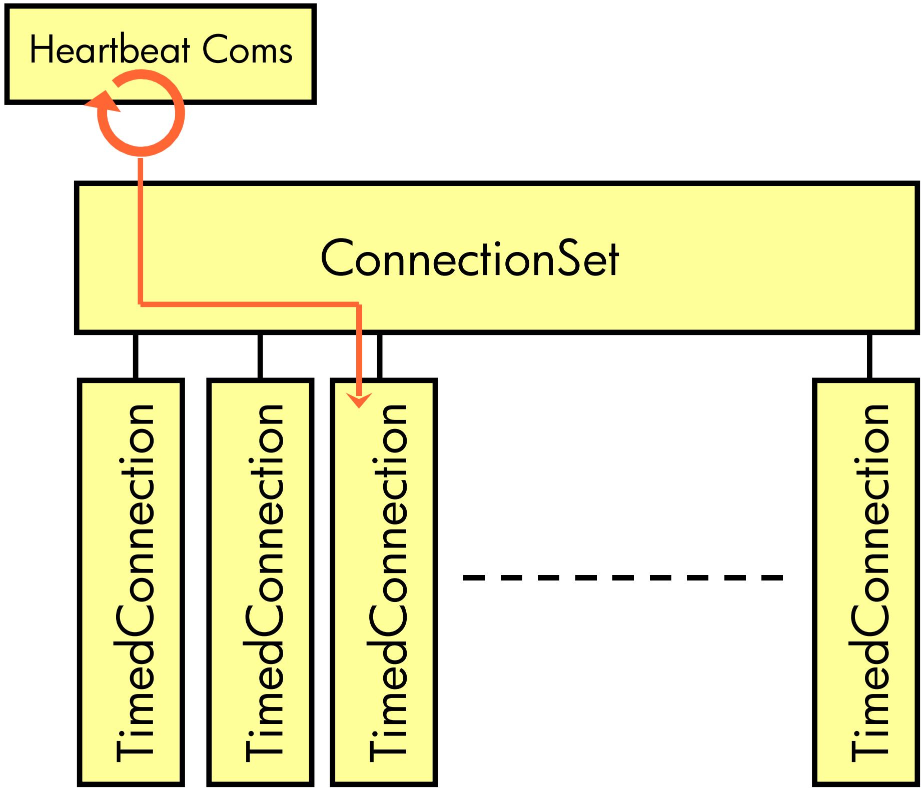
- The connection set maintains a view of timing:
  - The set of servers that are timely with respect to the local server
  - The connection set implements the local *connection view*
- Each timed connection is also a connection view
  - The connection view of the remote server
  - If a connection is timely its connection view is timely
- The local connection set is *consistent* when:
  - All *timely* connection views *agree* on membership
- The local connection view is *stable* when:
  - The connection set is consistent, and
  - The stability time is reached (see next bullet)
- The local connection view timestamp identifies its stability time
  - Initially, first moment of consistency + *stability interval*
  - Then adjusted to minimum of all timely connection view timestamps
  - Undefined when inconsistent

Connection Set



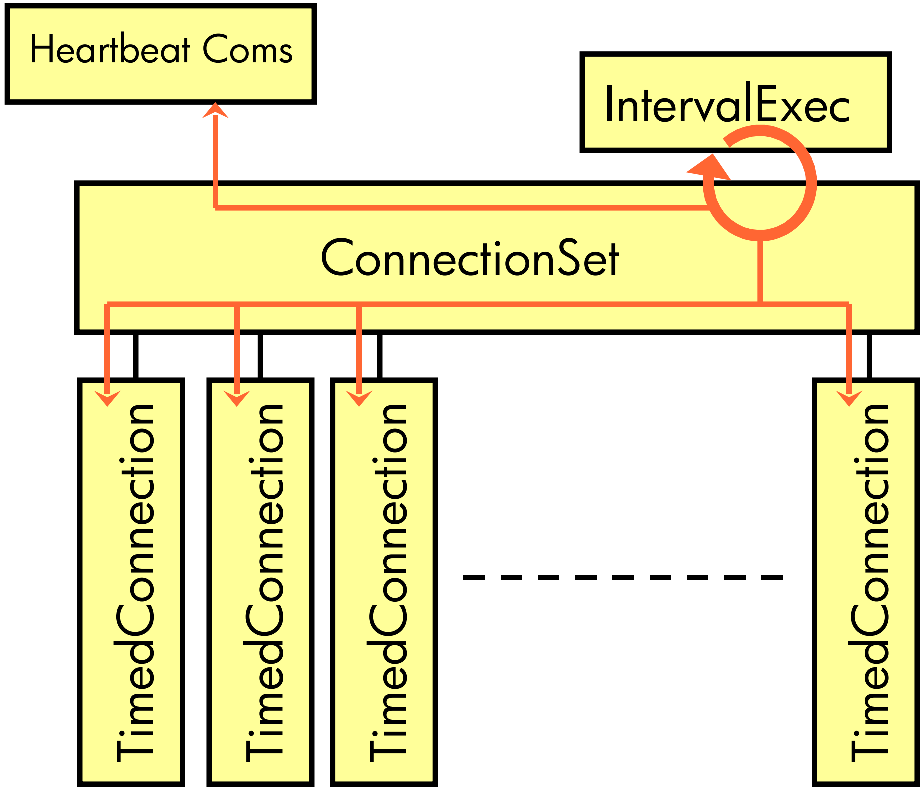
- ConnectionSet implements View
- ConnectionSet contains a collection of timed connections
  - These also implement View (HeartbeatProtocol extend View)
- The job of ConnectionSet is to implement the connection view
  - Identifies the *timely* connections
  - Identifies set *consistency*
  - Identifies view *stability*
  - Identifies the view *stability time*

Connection Set - Timeliness



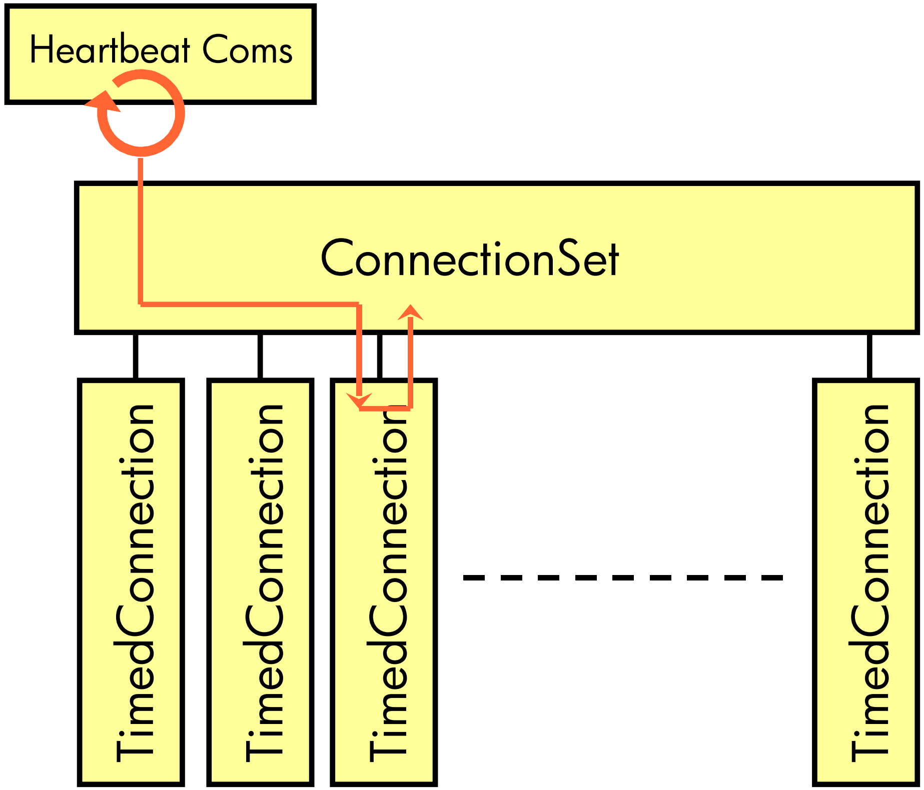
- Heartbeats are delivered to the appropriate connection by the connection set.
- If a corresponding connection does not exist, a new one is created
  - Heartbeat delivered to the new connection
  - If timely, added to the connection view

Connection Set - Timeliness



- IntervalExec orchestrates periodic operations
- Connections are periodically checked for timeliness and quiescence
  - Untimely connections are removed from the connection view
  - Quiesced connections are removed from the connection set
- A heartbeat is periodically sent by multicast

Connection Set - Stability



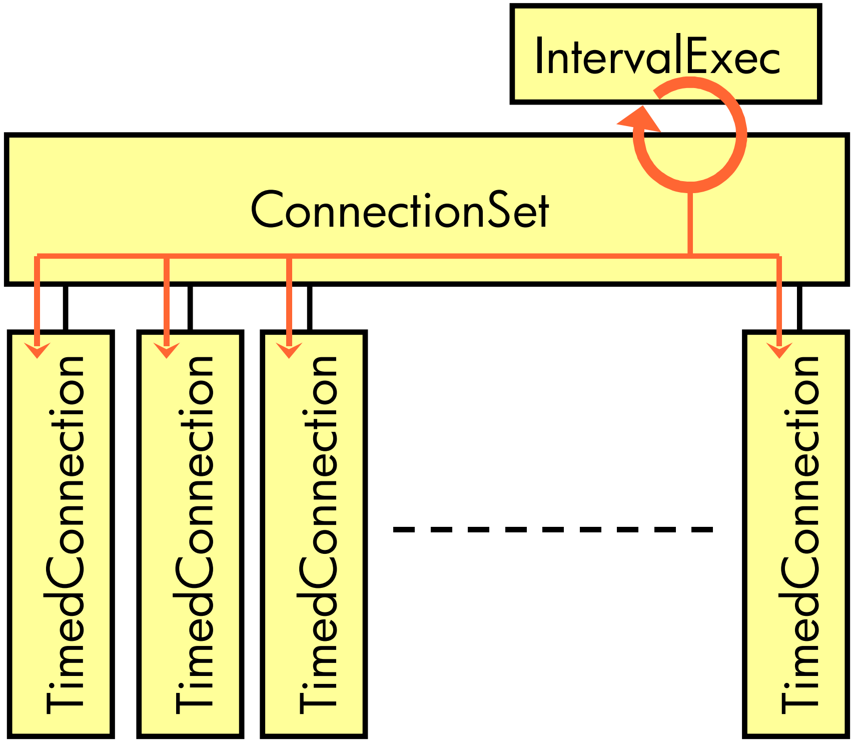
- Heartbeats contain the connection view of the sending server

- If a remote connection view changes, the heartbeat protocol of the timed connection will notify the ConnectionSet

- ConnectionSet will destabilise its connection view

- ConnectionSet is also notified if the timestamp of a remote connection set view changes

Connection Set - Stability

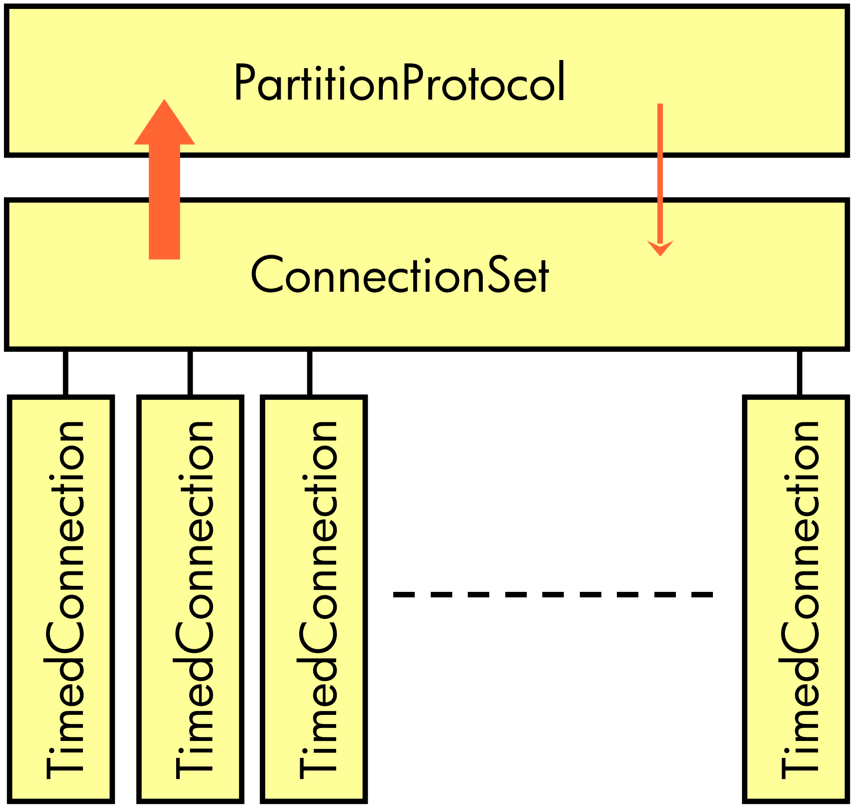


- When unstable, the connection set periodically checks for consistency
  - Less expensive than checking on every change
- If consistent, initiate a stability interval
  - Set timestamp to now + stability interval
  - or the earliest of all timestamps
- If new timestamps are reported the local timestamp may be adjusted

Partition Manager

- The partition protocol identifies the local partition view
  - It is a view
  - It implements the partition semantics
- The partition protocol operates as follows:
  - Members evicted if not timely (same as connection view)
  - Members evicted if they do not view local server as timely
  - Stability matches connection view stability
  - Members are only added when the connection view is stable
- The partition protocol also identifies the leader
  - Leader election on every partition change

Partition Manager



- ConnectionSet notifies PartitionProtocol when
  - Untimely connection
  - Lack of reciprocated timeliness
  - Unstable notification
  - Stable notification
- PartitionProtocol directs leader election on changes
  - Election protocol created by factory
  - Can be changed

Partition Manager

- PartitionManager provides:
  - registerPartitionNotification(PartitionNotification)
  - deregisterPartitionNotification(PartitionNotification)
  - Notification upcalls to:
    - PartitionNotification.partitionNotification(view, leader)

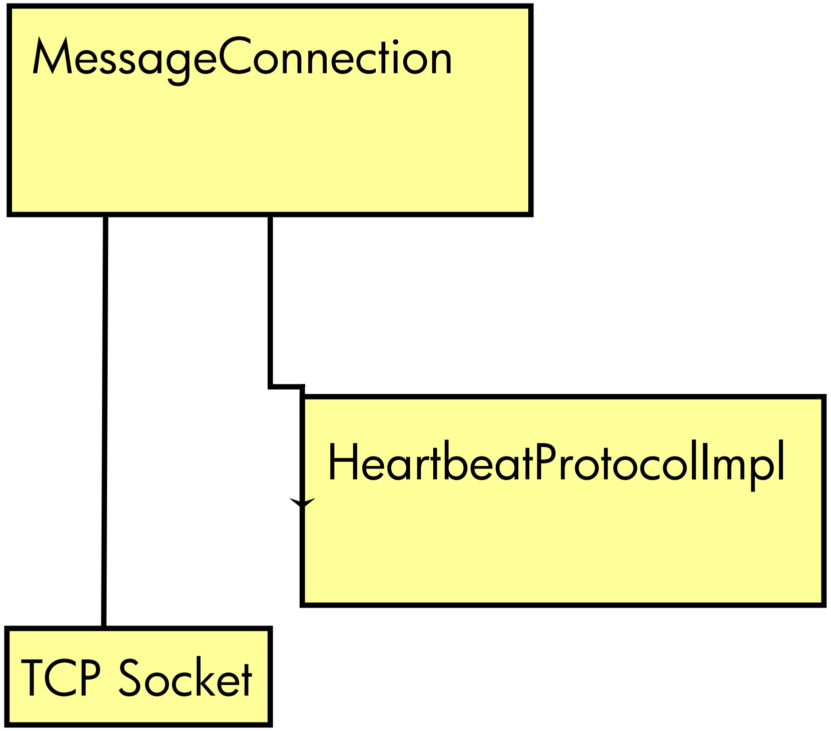
Timed Messages

- MessageConnection is an alternative implementation of timed connections that provides timely, ordered message delivery

- Connection-oriented protocol
  - Connections set-up/tear-down by switching HeartbeatConnection to MessageConnection and back
  - Connection set-up either end needs it
  - Connection tear-down neither end needs
  - All failures interpreted as timing failures (untimely)

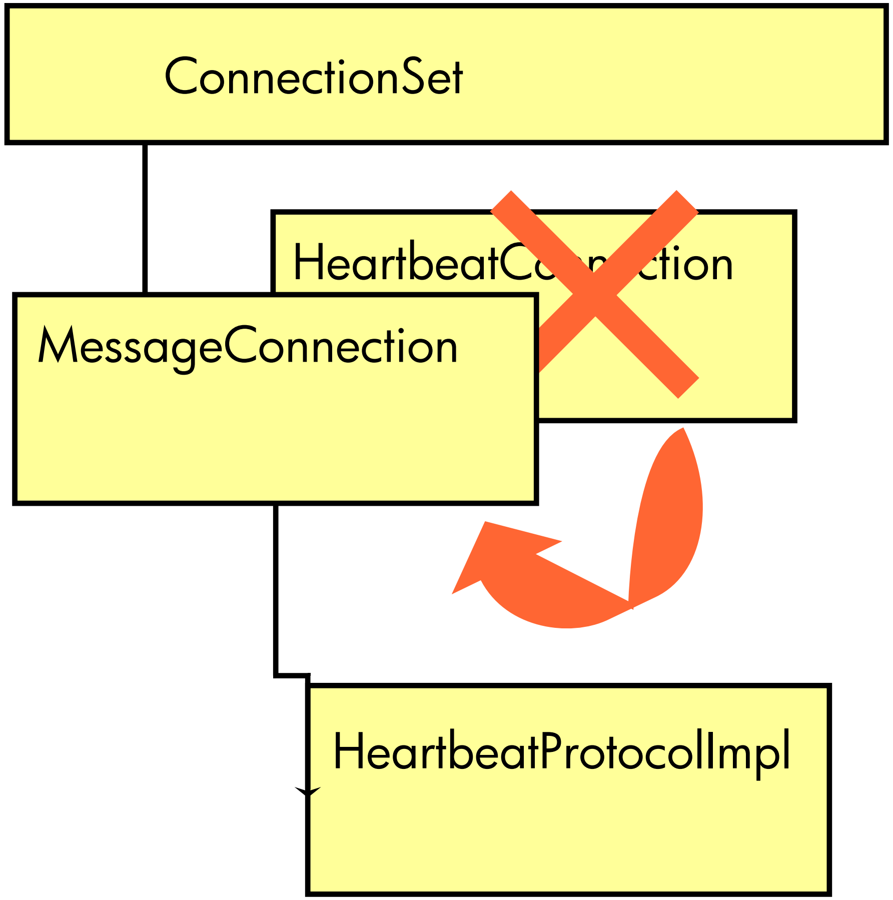
- The need for a connection is indicated in heartbeats
  - Same end always initiates TCP connect and disconnect
  - (the one with the lower id)

Message Connection



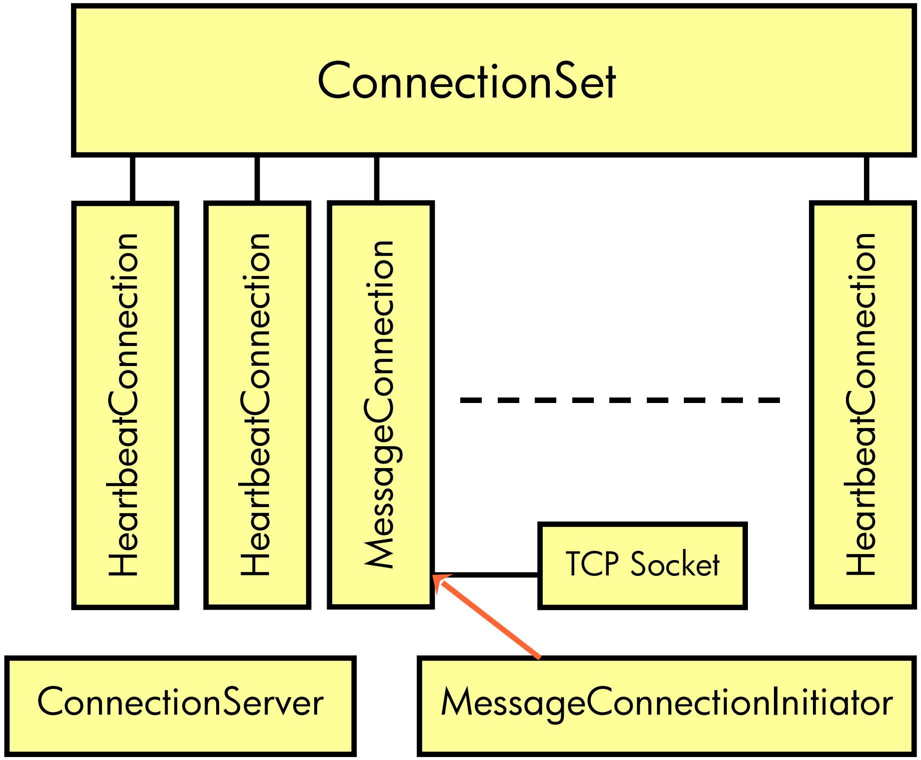
- MessageConnection extends HeartbeatProtocolAdapter
- Has own TCP socket connection
  - Receives heartbeats from TCP
  - Receives messages from TCP
  - All messages ordered
- Drops heatbeats that come from UDP multicast
- TCP failures lead to an untimely connection

Message Connection – Set-up



- Set-up is initiated at either end
  - Mark heartbeats
  - Only if timely
- ConnectionSet constructs MessageConnection passing the HeartbeatProtocolImpl from the old HeartbeatConnection
- MessageConnection immediately open for messages
  - Output messages buffered until they can be sent
- TCP connection follows set-up

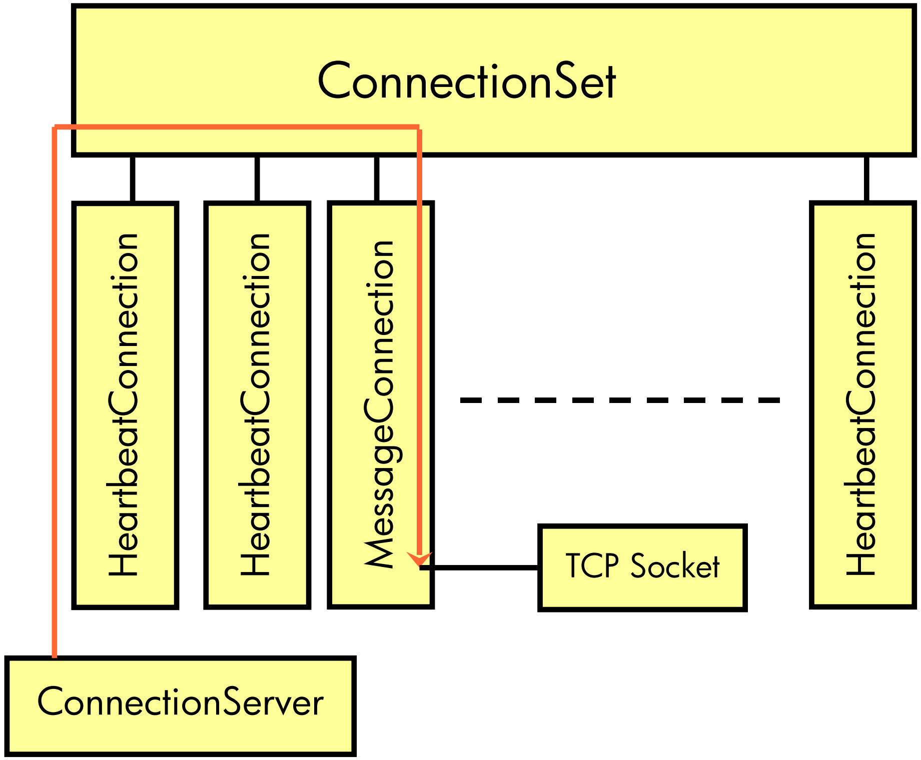
Message Connection – Set-up



If at TCP connect end

- Construct MessageConnection
  - At user request, or
  - When heartbeat marked
- Construction initiates asynchronous TCP Socket creation using MessageConnectionInitiator
  - TCP connect
  - Send initial heartbeat
  - Assign new socket back to MessageConnection

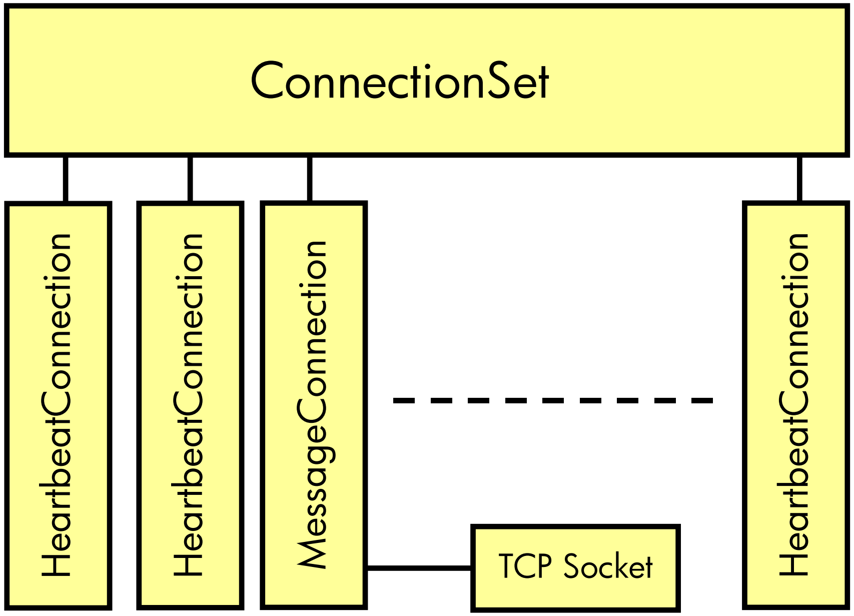
Message Connection – Set-up



If at TCP accept end

- Construct MessageConnection
  - If user requests
- Construction *does not* create TCP Socket
- ConnectionServer accepts connection and receives initial heartbeat
- If no MessageConnection, construct one with socket, otherwise just deliver socket

Message Connection – Set-up



Tear-down initiated by TCP connect end

- When no marking in heartbeats
  - Send close msg
  - Wait for close return
  - Revert to HeartbeatConnection
- Can be re-initiated at any point
  - E.g. due to new markings received during close

Partition Manager Interface

- PartitionManager provides:
  - registerPartitionNotification(PartitionNotification)
  - deregisterPartitionNotification(PartitionNotification)

- PartitionNotification has upcalls for:
  - partitionNotification(View partition, int leader)
  - objectNotification(Object object, int sender, long timestamp)

- To use the partition manager:
  - implement the PartitionNotification interface, and
  - Register

- You will be notified of:
  - partition and leader changes via partitionNotification(…)
  - Object delivery (messages) via objectNotification(…)

Partition Manager Interface

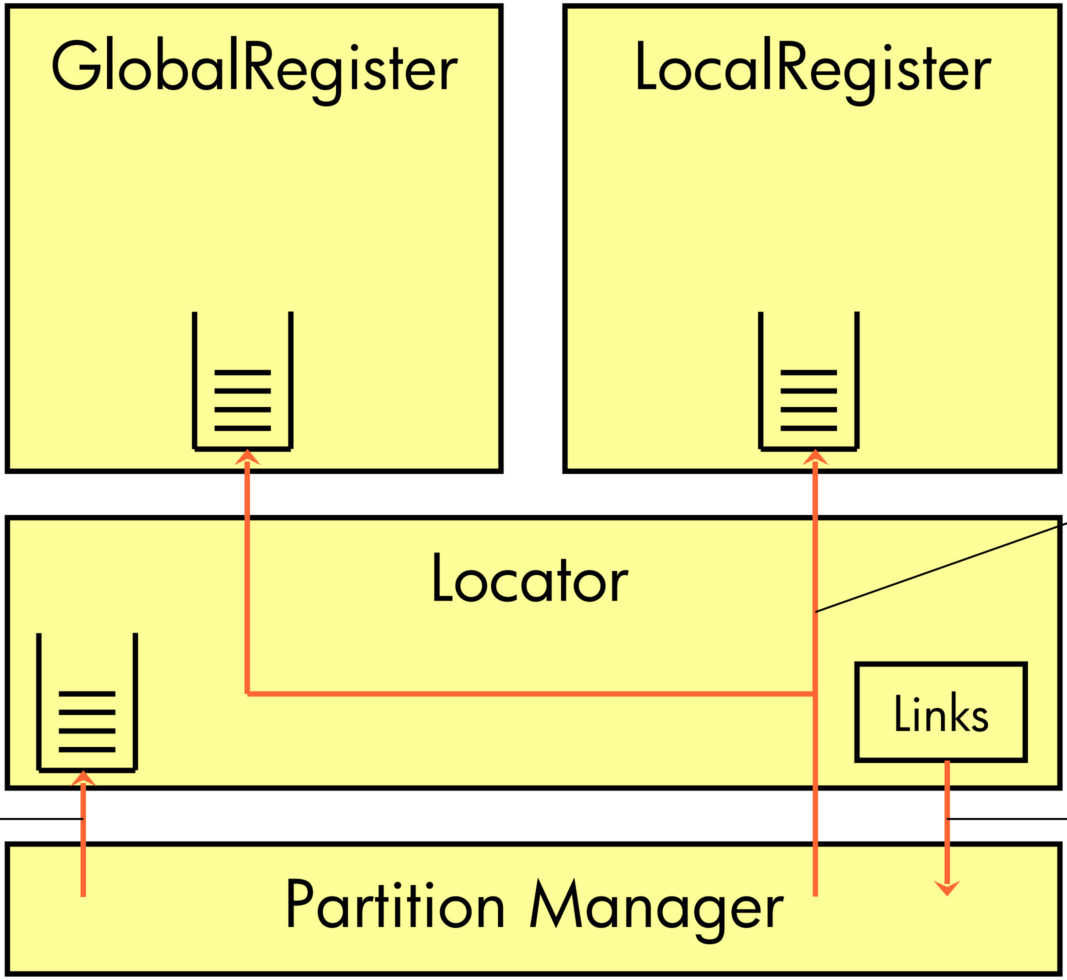
- PartitionManager also provides:
  - connect(node_id) – returns MessageConnection

- MessageConnection provides:
  - disconnect()
  - sendObject(object)

- MessageConnection can only be set-up between servers that have a mutual timeliness
  - If its not in the partition view, you can not send messages to it

- All MessageConnection failures are timing failures
  - Reported as a partition change!
  - If a new partition is reported, all missing connections have been closed, so stop using the handle

State Manager

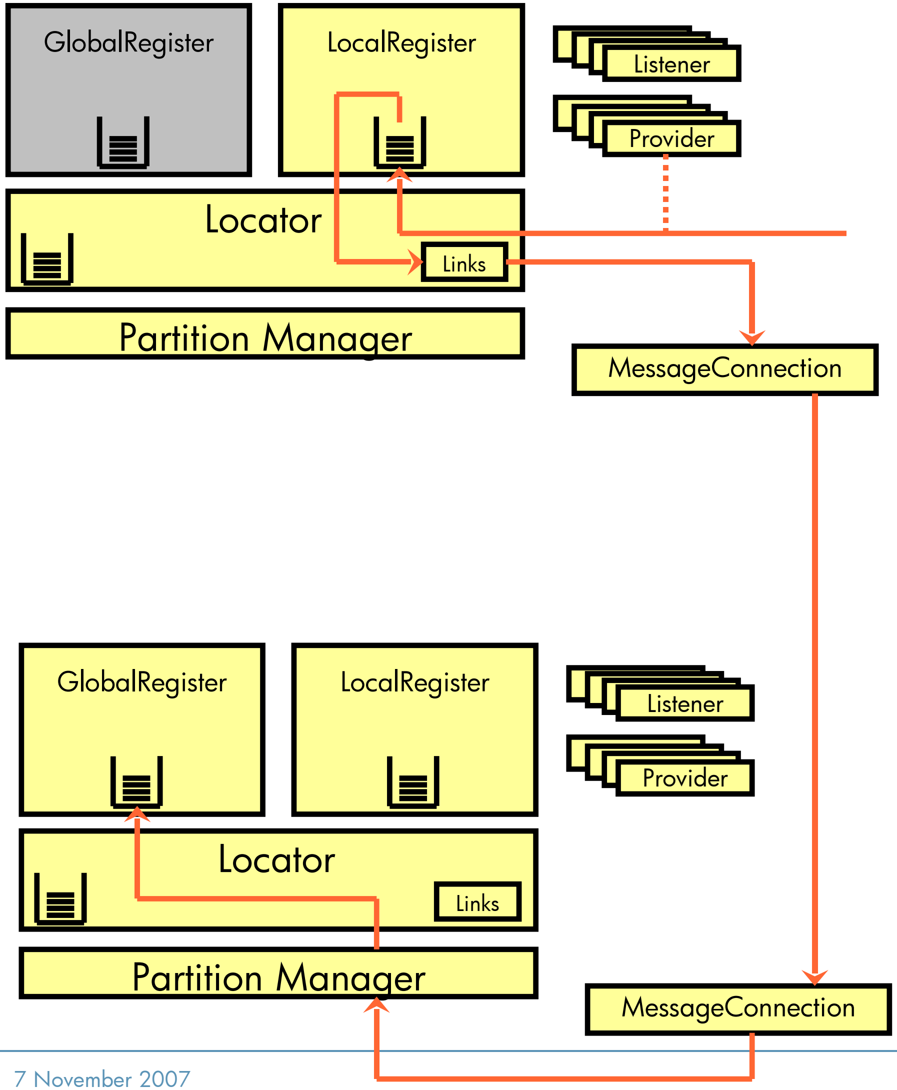


Object notifications routed to local / global register request queues

MessageConnections (links) created and removed as needed

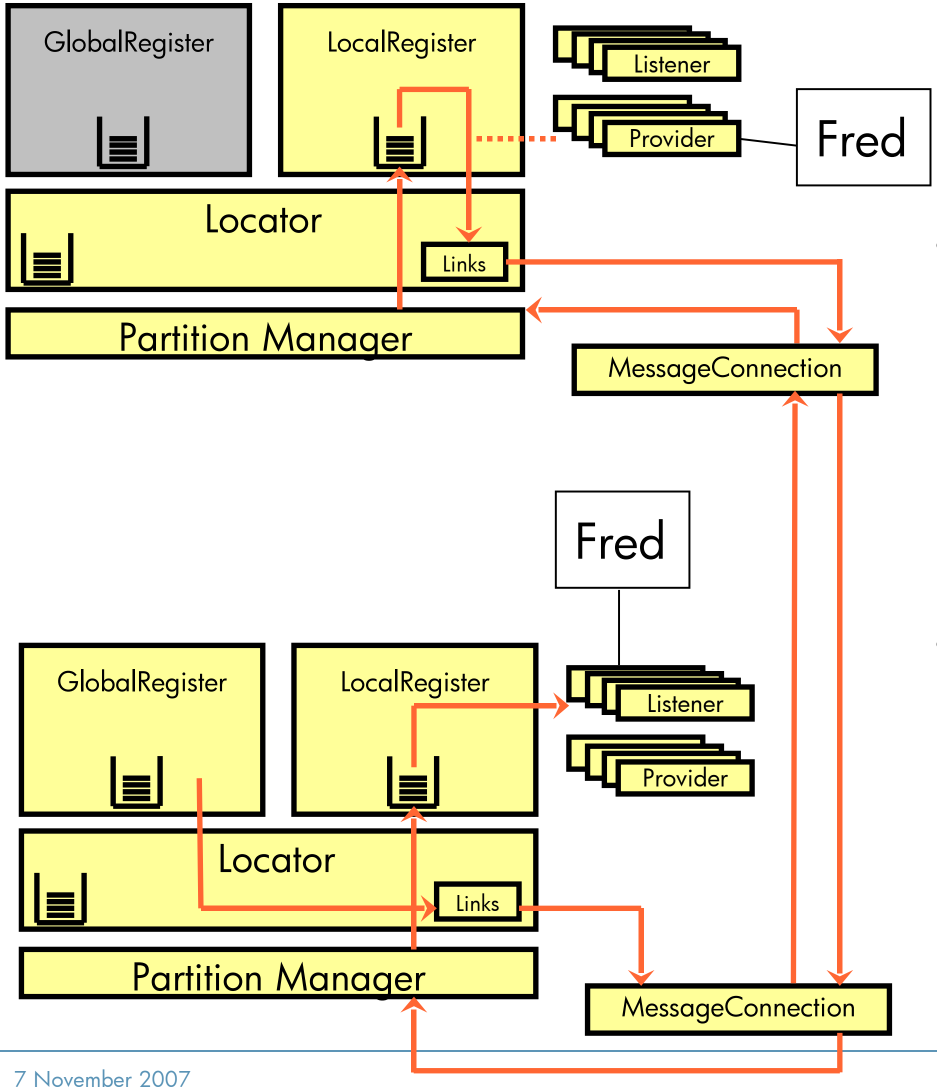
Partition notifications go to "stability queue"

State Manager – registration



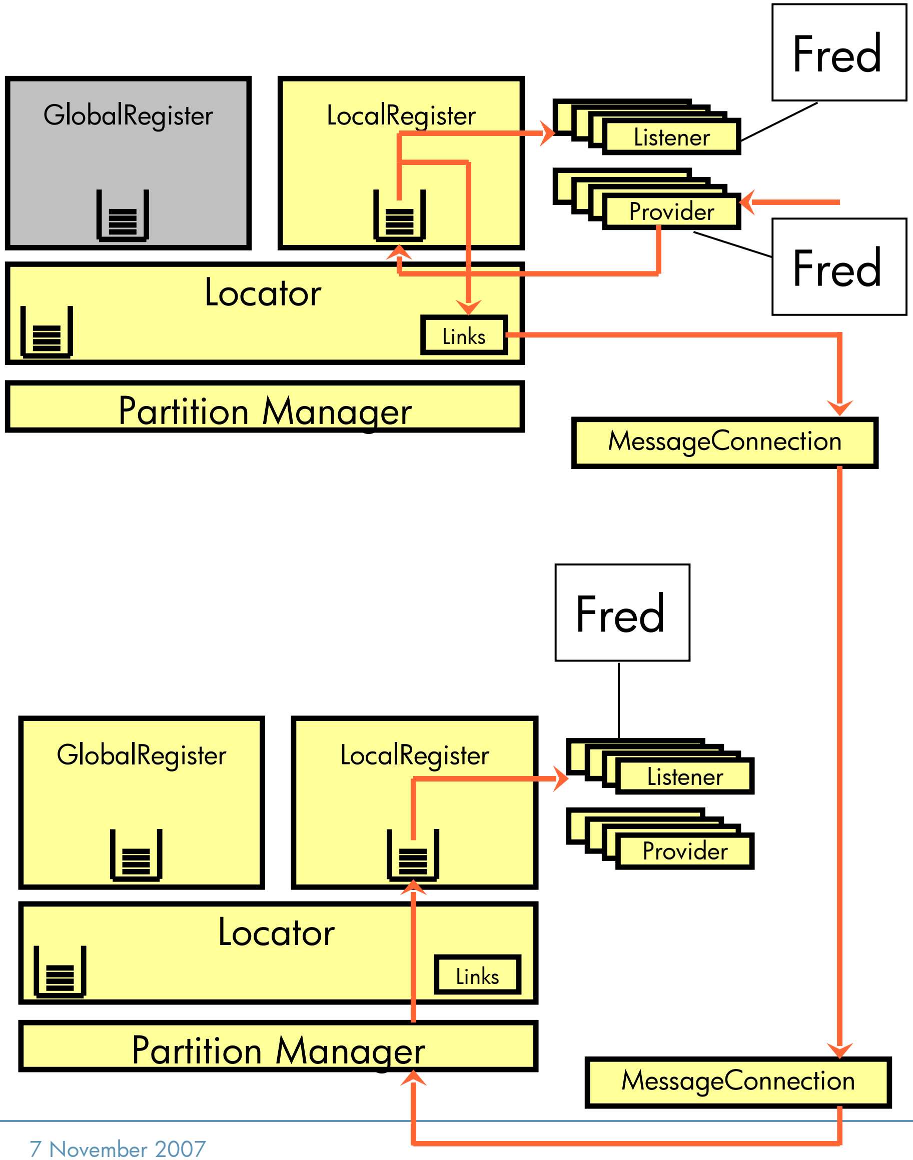
- New Provider or Listener is registered (locator interface)
  - Provider shown here
- Locator creates LocalRegister request
- If this is the first provider/listener with the given name at this server the LocalRegister informs the GlobalRegister
  - There is one global register
  - The server registers with the global register, not the listener/provider

State Manager – binding



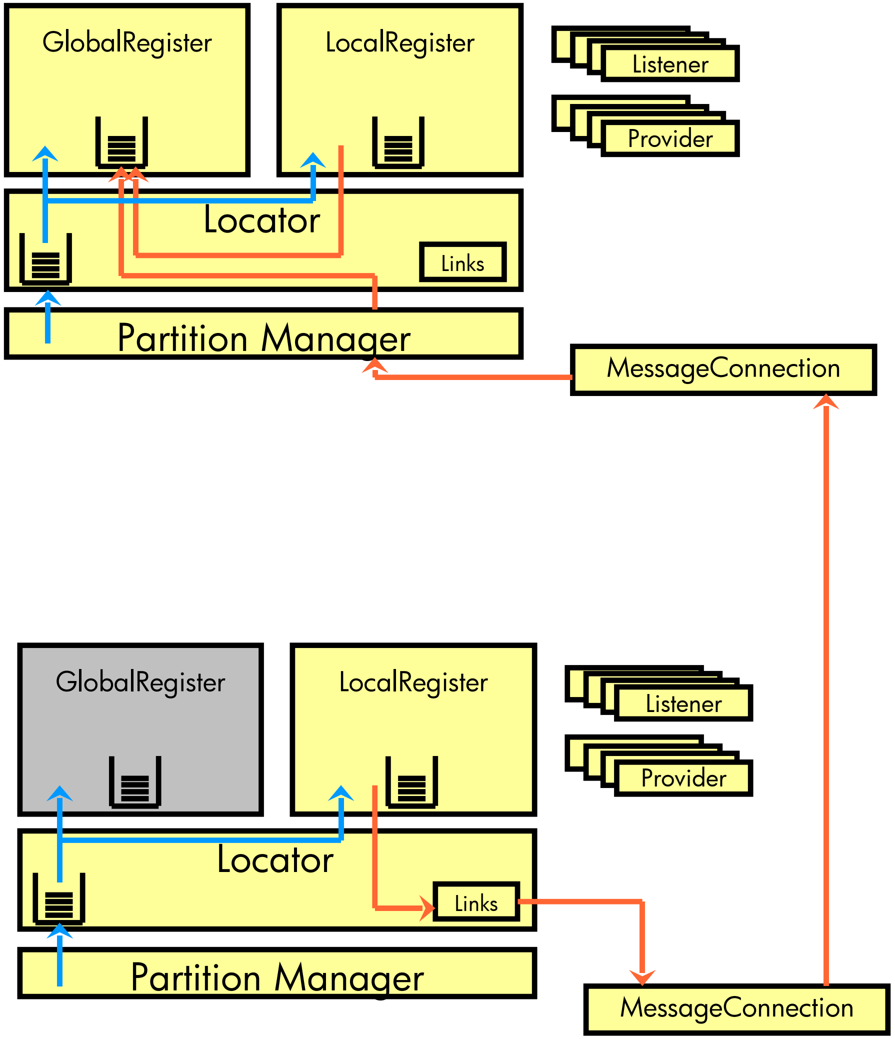
- If ever the global register identifies a new match between servers it informs *only* the server with the *provider(s)*
- The local register with the provider(s) notes the binding and sends updates to the server with the listener(s)
  - Binding record is ProxyListener
  - One update for each provider
- The local register with the listener(s) notes the binding and updates the listener(s)
  - Binding record is ProxyProvider

State Manager – delivering state values



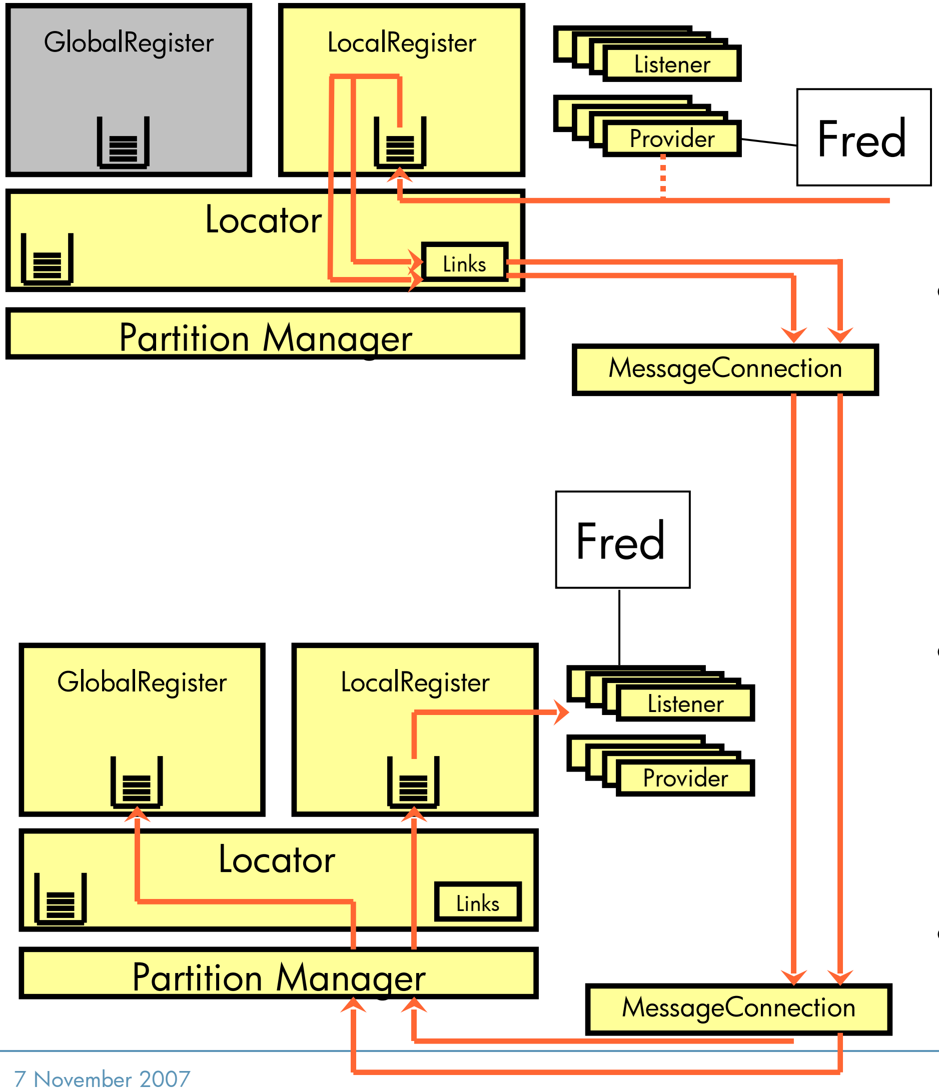
- User sets value on provider
  - Provider notifies locator
- Locator creates local register request
- Local register sends updates to:
  - local listeners, and
  - Bound remote servers
- Remote local register
  - Updates its local listeners

State Manager – global register failover



- When notified of new leader the partition will be unstable, immediately:
  - Non-leader clears its register
  - New leader resets its register
- When the partition stabilises
  - All servers re-register
- Guarantees global is available before it is used

State Manager – deregistration



- Provider or Listener is deregistered (locator interface)
  - Provider shown here
- If there are any bindings the local register sends deregister notifications to the servers
  - In provider and listener cases
- If this was the last provider / listener with the name, send deregister to the global register
- If a provider, local registers remove values from listeners

State Manager Interfaces

- AnubisLocator interface:
  - registerProvider(AnubisProvider provider)
  - deregisterProvider(AnubisProvider provider)
  - registerListener(AnubisListener listener)
  - deregisterListener(AnubisListener listener)
- AnubisProvider interface:
  - setValue(Object object) – must be serializable
- AnubisListener interface:
  - newValue(AnubisValue value)
  - removeValue(AnubisValue value)
  - createValue(ProviderInstance instance)
- AnubisValue interface:
  - newValue()
  - removeValue()

State Manager Interfaces

To use the state management part of Anubis

- Create providers by extending AnubisProvider
- Create listeners by extending AnubisListener
  - If you want you can also extend AnubisValue to create objects to deal with values – need to over-ride createValue(…) factory method

Tools

- You all know how to run Anubis
  - The sf files for the service and the tools are at the end of this presentation
- There are two debugging tools
  - AnubisPartitionManager
  - AnubisTestConsole
- AnubisPartitionManager
  - Provides a graphical view of what is going on in the partition protocol
- AnubisTestConsole
  - Provides the ability to observe the local server's state manager and dynamically create, update and remove providers and listeners

Anubis Partition Manager

- Start AnubisPartitionManager anywhere
  - It will snoop on multicast traffic to find servers
  - It will only see what is visible from its location
  - It must be set with the same multicast address and magic number etc.

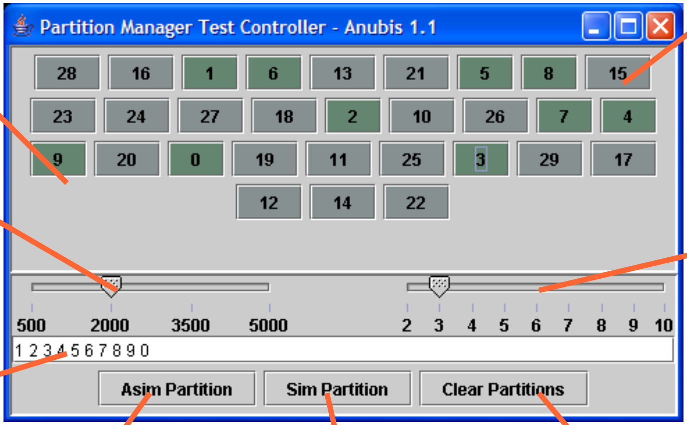
- Servers can be started with testable set to true or false
  - If true you get much more information from this tool
  - If false you only get connection view information

- This tool can be used to:
  - Fake network partitions
  - Observe the partition protocol
  - Check for real multicast network partitions and clock synchronisation problems
  - Tune the heartbeat timeout settings

Anubis Partition Manager

Main screen shows all visible nodes. If testable, these will be coloured according to their partiton

Black numbers = stable
Yellow numbers = unstable
Italic numbers = something wrong with time



Partition Manager Test Controller - Anubis 1.1

28	16	1	6	13	21	5	8	15
23	24	27	18	2	10	26	7	4
9	20	0	19	11	25	3	29	17
			12	14	22			

1 2 3 4 5 6 7 8 9 0

Slide-bar to control heartbeat interval

Slide-bar to control heartbeat timeout (multiple of interval)

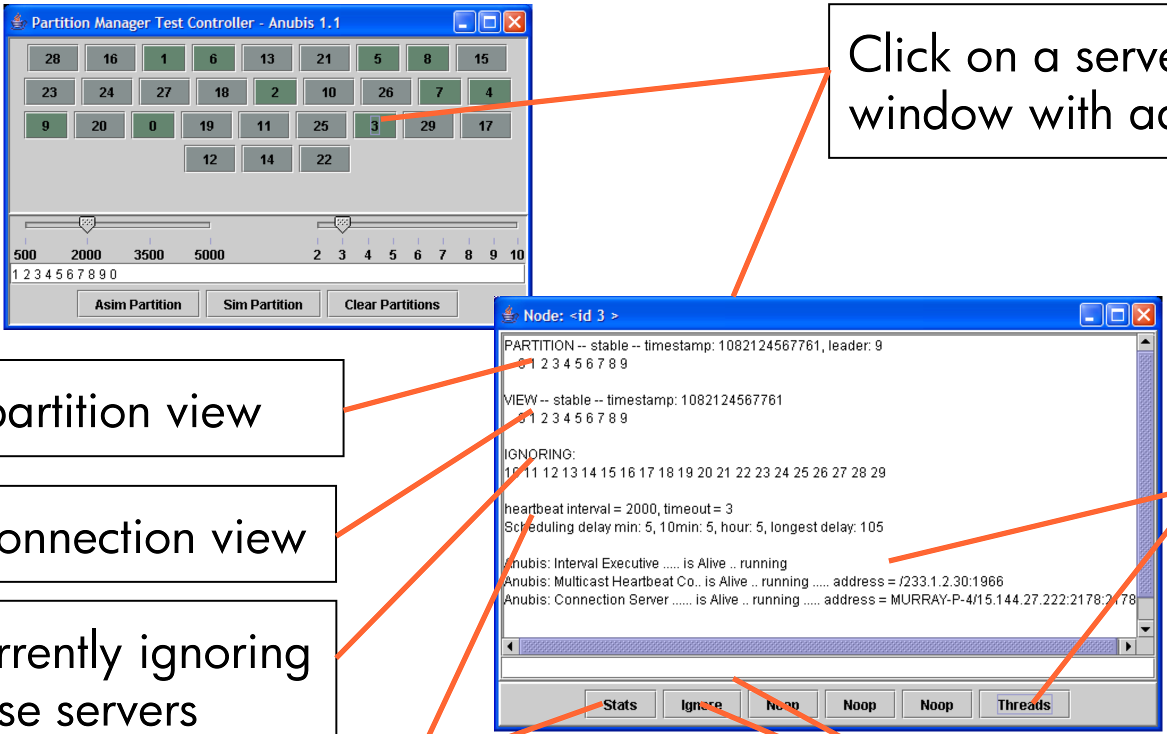
Input lists of servers here

Set asymetric partitons

Set symetric partitons

Clear partitions

Anubis Partition Manager



1 2 3 4 5 6 7 8 9 0

```
PARTITION -- stable -- timestamp: 1082124567761, leader: 9
  0 1 2 3 4 5 6 7 8 9

VIEW -- stable -- timestamp: 1082124567761
  0 1 2 3 4 5 6 7 8 9

IGNORING:
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

heartbeat interval = 2000, timeout = 3
Scheduling delay min: 5, 10min: 5, hour: 5, longest delay: 105

Anubis: Interval Executive ..... is Alive .. running
Anubis: Multicast Heartbeat Co.. is Alive .. running ..... address = /233.1.2.30:1966
Anubis: Connection Server ...... is Alive .. running ..... address = MURRAY-P-4/15.144.27.222:2178:2178
```

Click on a server button for a pop-up window with additional information

partition view

connection view

Currently ignoring these servers

Click on "Threads" to get information about main threads (remains for 10 seconds)

Timeouts and stats on scheduling delays refreshed periodically, click on "stats" to refresh immediately

Input a list of servers and click on "ignore" to set the ignored set of servers

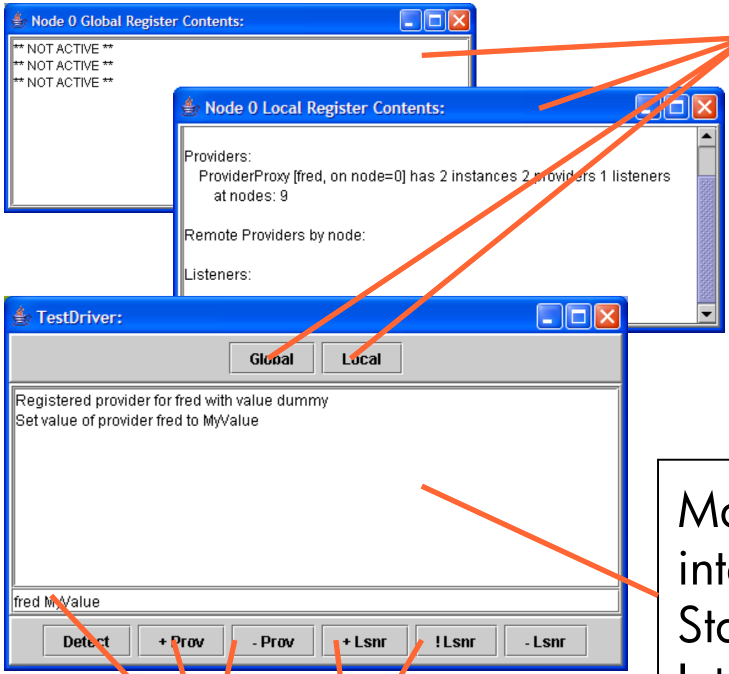
Anubis Test Console

- Start AnubisTestConsole in the same JVM as a server
  - It needs to find the locator interface (set in sf file)

- This tool can be used to:
  - Observe contents of global and local registers
  - Dynamically register and deregister providers and listeners
  - Set provider values

- Example
  - Start along side your application and view the states it creates
  - Inject state observations into an application by registering an appropriate provider
  - Observe specific states of your application by registering and appropriate listener

Anubis Test Console



Node 0 Global Register Contents:

```
** NOT ACTIVE **
** NOT ACTIVE **
** NOT ACTIVE **
```

Node 0 Local Register Contents:

```
Providers:
  ProviderProxy [fred, on node=0] has 2 instances 2 providers 1 listeners
    at nodes: 9

Remote Providers by node:

Listeners:
```

TestDriver:

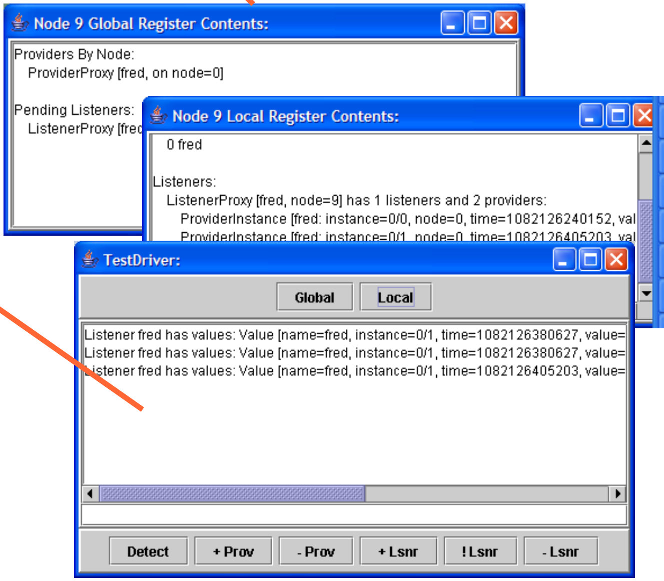
```
Registered provider for fred with value dummy
Set value of provider fred to MyValue
```

```
fred MyValue
```

Click on "Global" and "Local" to get pop-up windows on the registers

Only the leader has an active global register

Main screen shows interaction through State Manager Interfaces



Node 9 Global Register Contents:

```
Providers By Node:
  ProviderProxy [fred, on node=0]

Pending Listeners:
  ListenerProxy [fred
```

Node 9 Local Register Contents:

```
  0 fred

Listeners:
  ListenerProxy [fred, node=9] has 1 listeners and 2 providers:
    ProviderInstance [fred: instance=0/0, node=0, time=1082126240152, val
    ProviderInstance [fred: instance=0/1, node=0, time=1082126405203, val
```

TestDriver:

```
Listener fred has values: Value [name=fred, instance=0/1, time=1082126380627, value=
Listener fred has values: Value [name=fred, instance=0/1, time=1082126380627, value=
Listener fred has values: Value [name=fred, instance=0/1, time=1082126405203, value=
```

Enter a name and click on add (+) or remove (-) Provider to register/deregister providers – likewise for listeners.
(use <name> <valueString> to assign values)

Global Predicate Evaluation

I have not described the detector and do not intend to, except to say…

- Detector is a global predicate evaluator that can identify distributed states according to the consistency properties of Anubis Partitioned Knowledge Spaces
- DO NOT USE IT!
  - It does not correctly implement the semantics
- It is entirely above the Anubis State Manager interfaces and therefore does not affect the Anubis service

Example SmartFrog Files

```
#include "org/smartfrog/services/anubis/components.sf“

// deploys four AnubisServer components with an AnubisTestConsole
// component attatched to two of them (nodes 0 and 3)

TestNode extends AnubisService {
    testable "true";
}

sfConfig extends Compound {
    n0 extends TestNode { node 0; }
    n1 extends TestNode { node 1; }
    n2 extends TestNode { node 2; }
    n3 extends TestNode { node 3; }
    console0 extends AnubisTestConsole {
         locator LAZY ATTRIB n0:locator;
    }
    console3 extends AnubisTestConsole {
         locator LAZY ATTRIB n3:locator;
    }
}
```

```
#include "org/smartfrog/services/anubis/components.sf“

// deploys an AnbuisPartitionMonitor console

sfConfig extends AnubisPartitionTestConsole {
}
```

hp®

invent