Mobile Streams

M.Ranganathan♦, Laurent Andrey♦, Anurag Acharya♣, Virginie Schaal♦
mranga@snad.ncsl.nist.gov, andrey@snad.ncsl.nist.gov, acha@cs.ucsb.edu, schaal@snad.ncsl.nist.gov

♦National Inst of Standards and Technology
820 W. Diamond Ave
Gaithersburg, MD 20899, U.S.A.

♣Department of Computer Science
University of California
Santa-Barbara, CA 93106, U.S.A

Abstract

A large class of distributed testing, control and collaborative applications are reactive or event driven in nature. Such applications can be structured as a set of handlers that react to events and that in turn can trigger other events. We have developed an application building toolkit that facilitates development of such applications. Our system is based on the concept of Mobile Streams. Applications developed in our system are dynamically extensible and re-configurable and our system provides the application designer a means to control how the system can be extended and reconfigured. We describe our system model and implementation and compare our design to the design of other similar systems.

1. Introduction

A *Mobile Stream* (*MStream*) is a named communication end-point in a distributed system that can be moved from machine to machine as computation is in progress while maintaining a well-defined ordering guarantee. An *MStream* has a globally unique name and may be located on any machine that runs an execution environment for it and allows it to be moved there. Multiple event-handlers (*Handlers*) may be dynamically attached to (and detached from) an *MStream* and are independently and concurrently invoked for each message. *Handlers* operate in an atomic fashion. By "atomic" we mean that changes in the state of the *MStream* (which includes various attributes such as its location, the set of *Handlers* attached to it and so on) are deferred until the time when the *Handlers* complete execution.

The *Mstreams* model is based on the event-driven paradigm. An event-driven application is driven by asynchronous inputs that cause event-handlers to be invoked. A large class of distributed collaborative, testing, monitoring and control applications fit this paradigm - for example, conferencing and conference control applications, distributed control and testing applications [WM91], data combination [RAS98] and many others. In each case the notion of an "event" varies. In a collaborative system, events are user inputs; in a distributed monitoring and control system, events are changes in transducer inputs; in a distributed testing scenario, events are test outputs, timer alarms and so on.

In this paper, we describe the design and implementation of *AGNI*[1] – a *Tcl*-based multi-threaded system that supports the *MStreams* model. *AGNI* supports mobility and dynamic re-configuration – *MStreams* may be moved around during computation and new *Handlers* may be attached to and detached from *MStreams* dynamically. In contrast to *Agent Tcl* [G96] and *ARA[P98]* which support migration at arbitrary points in the program's execution – and therefore necessitate changes in the Tcl interpreter, all state changes such as *MStream* movements and new *Handler* registrations are deferred – they are effected only at handler boundaries. We call this the "mobile-server" model of mobility in contrast to the unrestricted model where migration can happen anywhere - which we call the "mobile thread" model of mobility.

AGNI is currently operational and has been used to implement three applications – a toolkit for collaboratively sharing Tk applications, a debugger for *MStream* programs and a monitor for visualizing an *MStreams*-based distributed system. In Section 2 we describe the *MStreams* programming model. In Section 3 we present simple code examples. In Section 4, we describe the applications that we have built with our system. In section 5, we describe the implementation of *AGNI*. In section 6 we compare *MStreams* with other Tcl and Java-based systems that support mobility. We conclude with a sketch of the direction in which we plan to take this work.

---

[1] Agents at NIST. Also the Sanskrit word for "fire".

2. Details

Our design separates name assignment of a communication end-point both from its location as well as from the functionality that is associated with it. So an *MStream* is simply a named communication end-point with an ordering guarantee and does not automatically imply a location wherein it resides. By adopting this model, we provide the flexibility of reconfiguring the system dynamically in two ways. First, since the name-location mapping is not specified a-priori, it can be moved dynamically during system execution. Second, since behavior is associated with a communication end-point rather than a location, it can be moved from location to location along with the communication end-point. An *Agent* is a collection of *Handlers* and includes an interpreter and a thread of execution. All the *Handlers* of an *Agent* share the same interpreter. A distributed system is organized around its communication end-points i.e. *MStreams*, and associating *Agents* with these end-points, which, in turn, attach *Handlers* for specific events. Several *Agents* may be associated with a given *MStream* with optional *initializers* and *finalizers*. An *initializer* is run when an *Agent* is first initialized at a given location. The *finalizer* runs at each location that the *Agent* has been initialized when an *Agent* is killed or exits. All of the *Handlers* specified by a given *Agent* share global state. An *Agent* may specify a portion of its global state as being in its *briefcase* – indicating to the system that this state needs to be re-located to the new location when an *MStream* is moved.

An *MStream* has a globally unique name. Logically, an *MStream* has two "ports"– a data and a control port; and two "event-types" – data and control events. The data port is the default port to which messages destined for the *MStream* are sent. Control events are delivered to the control port. A data event is triggered by a message delivery to an *MStream* and control events are triggered by the control actions that we describe below.

An *Agent* may register an *append-handler* for data events and a *relocation-handler* for the *relocation* control event. Multiple Handlers may be associated with each of these events; a given *Agent* may register at most one handler for a given event type. *Handlers* are run when the corresponding events occur. At message delivery, each of the *append-handlers* is run concurrently and when the *MStream* is relocated from one location to another, each of the *relocation-handlers* is run concurrently at the target location. *Handlers* may append messages to other *MStreams*.

Each *MStream* has a *Stream Controller*. The *Stream Controller* is a privileged *Agent* and can register handlers for various control event types. Control events logically fit into two classes: (1) events that are triggered as a result of an attempted re-configuration or extension of the system and (2) events that are triggered as a result of data being appended to the *MStream*. There are 8 types of control-events. The corresponding handlers are: (1) a *New-agent handler* that can decide whether or not to allow an *Agent Attach* operation to succeed on the *MStream*; (2) an *Agent-kill handler* that can decide whether or not to allow an externally initiated destruction of an *Agent* to succeed (3) a *Stream-move handler* that can decide whether or not to allow a relocation of the *MStream* to succeed; (4) a *pre-arrival handler* that gets control at the target location of the relocation before the data event-stream *relocation-handlers* get control[2]; (5) a *Stream-open handler* that can decide whether or not an *open* operation succeeds on the *MStream*; (6) an "out of band" (OOB) *append-handler* that runs when a *control-append* is processed ; (7) a *pre-append handler* that runs prior to any of the data event-stream *Handlers* getting control; (8) a *post-processing* handler that gets control after any agent-registered handler runs. The pre-processing handlers may modify the execution environment of the data event-stream handlers and may execute commands in the execution environment of any of the handlers.

Data sent to the data port is usually seen by all non-privileged *append-handlers* registered with the *MStream*. Data sent to the control port is only seen by the *OOB append-handler*. Data *appends* can be blocked by the pre-append *Handler* registered by the *Stream Controller*. As an example presented later in this paper will show, blocking can be useful for synchronizing actions. OOB appends cannot be blocked. All *append* messages, control and data are consumed in the same order in as the order in which they were sent; however, there is no ordering guarantee between messages sent to the data port and those sent to the control port. The message ordering guarantee is preserved even when both sender and receiver are in motion. Thus, if an *Agent* of an *MStream* ***a*** from location ***x*** *appends* a message "1" to an *MStream* ***b*** then goes over to location ***y*** and *appends* message "2" to stream ***b***, the *Agents* of ***b*** should consume the messages in the order <"1","2">.

Figure 1 shows what happens in the system during the common case of *data append* processing. First, the *on_append* pre-processing handler gets control. This handler has the responsibility of scheduling the *on_append* handlers registered by the Agents of the *MStream*. It achieves this by selecting the handlers to run and marking them "runnable".

---

[2] Note that an *Agent* may also register a handler for the relocation control event. However, the pre-arrival handler gets control before the *Agent* relocation handler gets to run

After the *Agent*-registered *on_append* handlers have completed execution, the *Stream Controller* registered *post-processing handler* gets control. Relocation processing follows along the same lines with the pre-processing handler getting control on arrival at the new location and marking the other handlers *runnable* before they get to execute and the post-processing handler getting control after the selected handlers have completed execution.

System Specific Events	Location Specific Events	Stream Specific Events	
		Data	Control
			New Agent
Stream Create	Arrival	Append	Agent Kill
			Stream Move
			Pre-arrival
New Peer			Arrival
			Stream Open
			OOB Append
Stream Destroy			Pre-append
			Post-process

Table 1 : Event categories and types

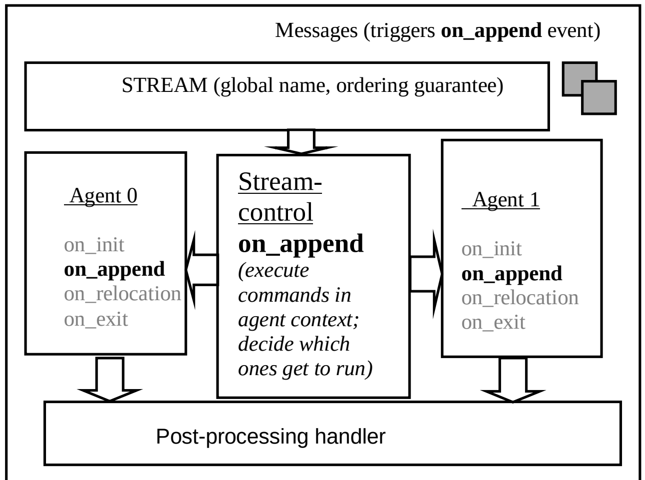

Figure 1: Append processing.

The stream-controller has the opportunity to intervene before the Agent-registered Handlers get to execute and after agent-registered Handler execution has completed.

Each workstation that wishes to participate in the system runs an *Agent Daemon* that has a unique location identifier. Each *Agent Daemon* may be started with a *Site-Controller* script specified. The *Site-controller* may register a handler that is invoked on *MStream* entry into the location. It may approve or deny the attempt to enter by returning an appropriate return-code and may restrict access to resources such as files at the site. A specific *Agent Daemon* is designated the *System Controller* and can include a *Handlers* for three "system events": (1) "new-stream" event-handler that is consulted when a new *MStream* is created (2) a "new-peer" event handler that is consulted when a new *Agent Daemon* attempts to join the system. It may approve or deny the join attempt by returning an appropriate error-code and (3) A s*tream-destroy* event handler that is consulted when an *MStream* is destroyed. Table 1 shows the various event types.

3. Examples

In this section we provide two introductory examples to illustrate our programming model. More detailed applications of our system are presented in the next section.

Consider a networked chat application that works by each participant broadcasting messages to all other participants. In such an application, it may be desired that all participants see an evolving conversation in the same order. Such an application may be constructed using *AGNI* by sending the input from each participant to an *MStream* which then re-broadcasts events to all participants – thereby ensuring that each participant sees the events in the same total order. The *MStream* that re-broadcasts events may be periodically re-positioned to reduce expected latency. This is actually a skeletal version of the Tk collaborative toolkit application that we present in the next section and is representative of an important class of distributed systems whose design can be simplified by imposing a global ordering constraint on messages. Figure 2 shows the logical structure of the application and Figure 3 the agent code for the *MStream* that re-distributes events.

Our second example illustrates the use of the control part of a *Stream* for synchronization. Assume we wanted build an application using two *MStreams* such that their inputs will be consumed in a synchronized fashion. This can be achieved by using a *MStream* controller and a *control-append-handler*. When an *Append* arrives at each stream, the input is blocked by the append pre-processor (registered by the *Stream Controller*). A notification is sent to the synchronizer stream. When the synchronizer receives input from both streams, it sends notification for them to proceed by sending a control append to each one. The application structure is shown in Figure 4 and its *Agent* code is included in the appendix.

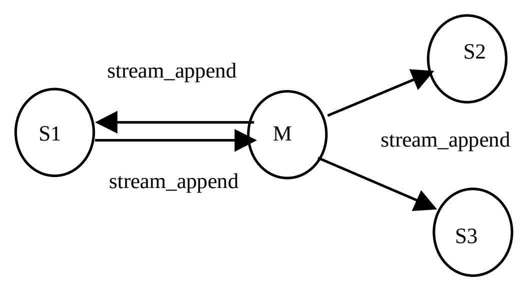

Figure 2: Structure of chat application

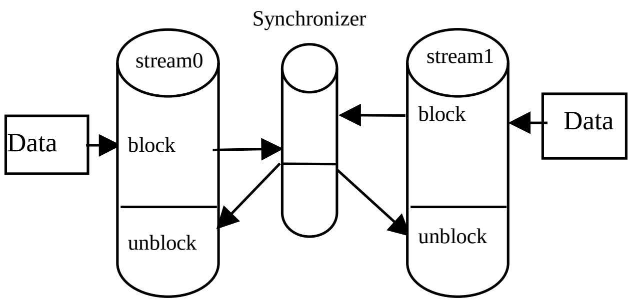

Figure 4: Logical structure of the stream synchronizer.

```
stream_open  M
set consumers [list S1 S2 S3]
register_agent  M on_init  $consumers  {
        global counter
        global  consumers
        set consumers $argv
        foreach consumer $consumers {
              set counter($consumer) 0
        }
        set count 0
} on_append {
          incr count
          set  loc [stream_sender_name]
          incr counter($loc)
          foreach stream $consumers {
                 stream_append $stream $argv
           }
           if  { $count  == 50 } {
                #find_max finds the max counter value
                set  newloc [ find_max counter]
                stream_relocate  [stream_location $newloc]
           }
} on_relocation {
         foreach consumer $consumers {
              set counter($consumer) 0
        }
        set count 0
 }
```

Figure 3: Chat application agent.

Note that we have made no assumptions about the structure or function of the other *Agents* that are associated with *stream0* and *stream1*. We have just imposed the constraint on the streams that ensures that the stream0 and stream1 will consume inputs in a synchronized fashion. The synchronizer and both the *MStream*s can be mobile.

4. Applications

In this section, we give an overview of three applications, the first of which is a toolkit for building collaborative Tk applications; the other two being "system level" applications. Since our purpose is to provide an overview, we describe the applications in brief, providing highlights of the implementation.

4.1 Collaborative Sharing of Unmodified Tk Applications

Our first application example is a toolkit for sharing unmodified Tk applications. The purpose of the toolkit is to provide a mechanism to collaboratively share arbitrary Tcl/Tk applications using the “What You See is What I See” (WYSIWIS) paradigm. Each user runs a separate copy of the application. What each user inputs to her GUI must be replayed on every other user’s copy of the application so that the WYSIWIS guarantee may be preserved. The application may be sensitive to the order of input actions and hence, the input events must be replayed in the same order on each user’s copy of the application.

Our system works by “rebinding” each Tk widget. The rebinding code (using the approach as taken by Tk Replay [Cr95]) visits all the widgets in the widget hierarchy and finds the tags bound to it. Then it finds each event that is bound to the tag and rebinds it. For each binding, the original script is saved in a table and replaced by a binding that sends the action to a central “event re-dispatcher” *MStream* using an *append*. When the user performs an action on the GUI the event is dispatched to the “event re-dispatcher” MStream - bypassing the binding script that would normally get called. The re-dispatcher MStream resends the event to each participant by appending to a stationary MStream located on each participant’s machine. When the dispatcher sends actions back to the application, the original binding script is replayed. These events are then sent to the application via a socket. The application executes the Tk commands it receives via the socket.

If the central event re-dispatcher were stationary, this could become irritating for the user actually manipulating the interface as she would experience a round-trip latency for each input event. We circumvent this problem by dynamically repositioning the re-dispatcher. The re-dispatcher keeps a count of source Tk Events and moves to the source location when the count exceeds a threshold – thereby reducing the expected latency for the interactive user. The *MStream* ordering guarantee ensures that order is maintained despite this re-positioning – thereby ensuring the correctness of the application. The application structure is as shown in Figure 5 and a screen shot of one of the applications that we shared using this approach is shown in Figure 6. Our implementation of the Tk-collaborative toolkit works well at threshold counts of about 30 to 50 events. The improvement in performance due to latency reduction is to be balanced against the protocol overhead cost caused by frequent re-positioning of the dispatcher.

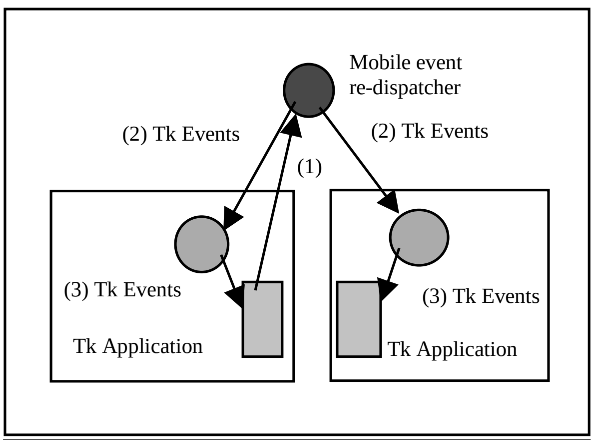

Figure 5: Tk- Collaborative Toolkit Architecture Events are sent to the dispatcher stream first and from there to each participant.

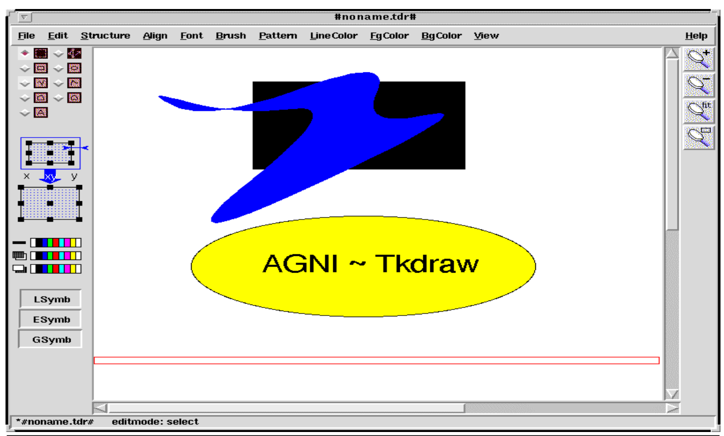

Figure 6: Screen-shot of an unmodified freely available drawing program (Tk Draw [T98]) that was shared using the application sharing toolkit.

4.2 A Monitor for Visualizing MStreams

The *MStream Monitor* provides several services. First, it provides a graphical view of the system – indicating where the *MStreams* reside. Second, it provides an interface whereby users can effect actions such as creating new streams, killing *Agents*, attaching new *Agents* and so on. Third, it provides an interface to an *Agent Debugger* that we describe in the next section. Figure 7 shows the logical organization of the *Agent* monitor and the actions that happen on an *MStream* relocation. Figure 8 shows a screen-shot of the monitor GUI. The Monitor GUI receives commands to update its appearance in reaction to state changes in the system via a *monitor Mstream*. The Site-controller on each workstation sends the *monitor MStream* notification on new *MStream* arrivals. The system controller sends the monitor *MStream* notification on new *MStream* and new *Agent Daemon* creations. The on-append handler of the *monitor MStream* ships the appropriate Tk commands to the monitor GUI to update its appearance.

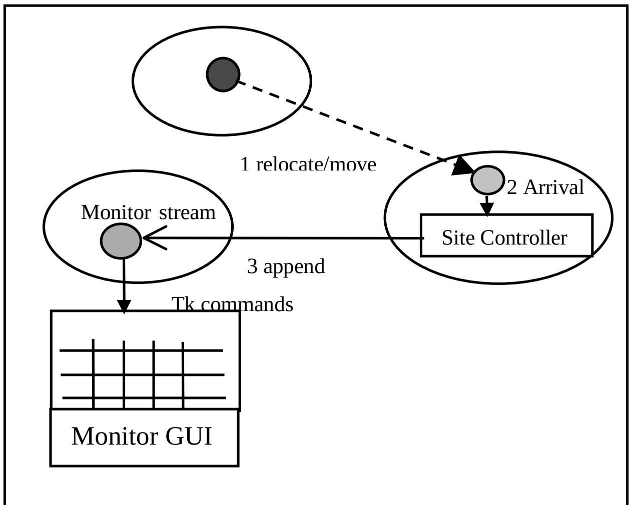

Figure 7: Organization of the Agent Monitor

The monitor GUI reacts to system changes by updating its appearance. Chain of events that occurs on a move is shown.

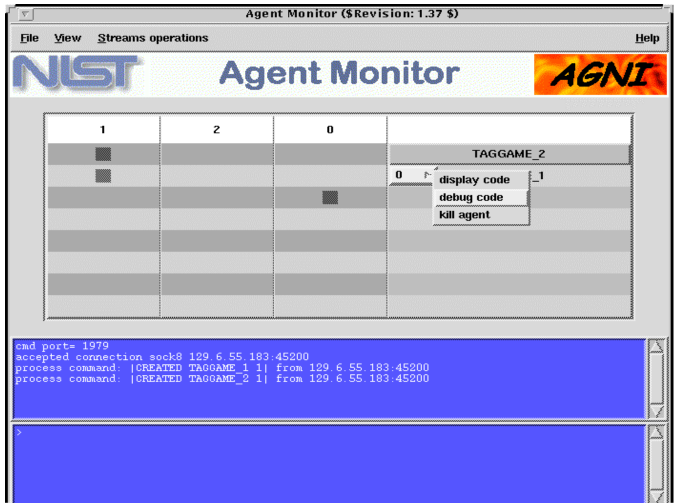

Figure 8: Agent Monitor GUI

Allows users to visualize and change the state of the system. The Monitor works as a reactive application – it receives inputs via the *monitor MStream*.

At any given time, several *Agent* monitors may be simultaneously active. Control requests on an *MStream* are processed in FIFO order at the location of the *MStream*.

4.3 A Distributed Debugger for MStreams

One of the arguments in favor of mobility is that it simplifies the construction of distributed systems. However, debugging a mobile program can be difficult. The debugger has to be transparent to the application program and has to carry state information as the program moves around. It also has to redirect *stdin* and *stdout* to a well-known place as the program moves around. Finally, the debugger must debug the mobile program in a location-transparent fashion – a *Stream* move should not disrupt the debugging session.

We have developed a debugger for our system that permits position independent debugging of *MStreams* programs. The debugger allows users to set watch-points on *Agents*, single-step the code of the *Handler* and so on. Our debugger supports global watch-points. A global watch-expression may include sub-expressions that are evaluated in the context of *Agents* of different *MStreams*. When the global watch-expression evaluates to TRUE, debuggers may be started on the *Agents* that own terms participating in the watch-expression. In keeping with the design of the rest of the system, global watch-points are only evaluated at handler boundaries.

To debug a stream, it must be started with a special debugging *MStream Controller* – which contains the state for the debugger. The debugger code is placed in the pre and post-processing control-handlers (registered by the stream-controller). The debugger gets control before any *Handler* gets to run and after the execution of the *Handler* has completed via the pre and post-processing handlers. While the agent-registered handler is executing under debugger control, its "debug state" may be changed by inserting watch-points etc. This state is actually stored in the debugee's interpreter. If the stream is moved to a new location, the watch-points have to be installed in the environment of the target location. Since movement is restricted to occur only at the end of a handler's execution, watch-points for each *Agent* can be retrieved by the post-processing control handler prior to *MStream* movement. At the target location, the watch-points may be re-installed during on-arrival control pre-processing (before the on-arrival data *Handlers* get to run). The control handler thus manages state needed by the debugger in a transparent fashion. Global watch-expressions also evaluated using *MStreams*. The first step in evaluating a global watch-expression is for the *Stream Controller* of each of the Streams whose Agents "own" terms belonging to the expression to extract the value of these terms. These values are sent to a global watch-expression evaluator after blocking the stream. If the expression evaluates to TRUE, the global watch-expression evaluator returns this notification, causing the debugger to gain control. In our system, *stdin* and *stdout* are directed to the debugger console via *MStreams* as shown in Figure 9 below. The debugger itself was constructed using an almost unmodified version of a freely available Tcl debugger [L93].

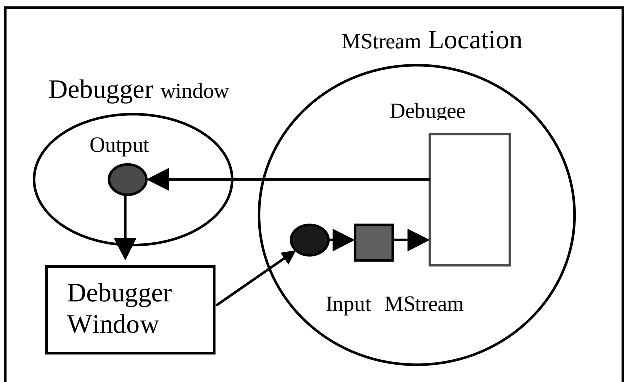

Figure 9: Organization of Streams for interaction with the debugger

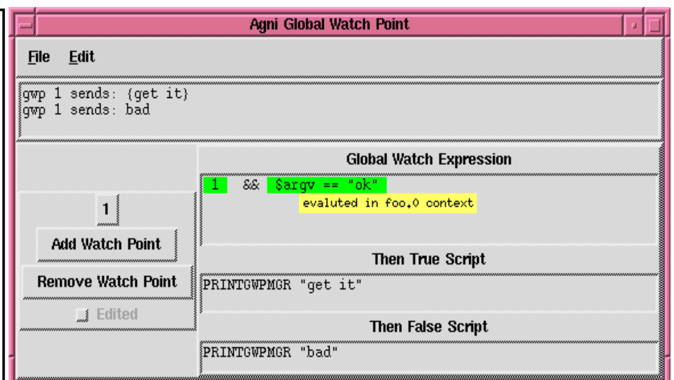

Figure 10: Global watch-point expression window.

5. The Implementation of AGNI

Each workstation that wishes to participate in the distributed system runs a copy of an *Agent Daemon*. Each Agent Daemon has a unique identifier. A distinguished Agent Daemon is designated the *System Controller* and is in charge of accepting or rejecting new *Agent Daemons* and also serves as a directory manager; it is informed when the location of each *MStream* changes. Each of the other *Agent Daemons* maintains a connection with the *System Controller Agent Daemon*. Conceptually, the arrangement is as shown in figure 11.

Except for the case when the MStream is co-located with the location from where the message originates, all control events destined for an MStream are directed through the *System Controller* via the TCP connection that each Agent Daemon maintains with the System Controller. The System Controller also acts as a directory manager – keeping track of where each *Stream* is located and is hence able to re-direct control message to the location of the *MStream.* Sending all control events through the system controller is a simple means of achieving a global ordering on control events. The negative aspect of this design is that the System Controller has the potential of becoming a bottleneck. However, we expect the number of "control events" to be much smaller than the number of "data events" processed by the *MStream*.

Data events are delivered to the *MStream* directly without going through the *System Controller*. Data messages are delivered asynchronously with control returning to the sender immediately. In the meanwhile, the system may be reconfigured by moving the target *MStream* to a new location. Messages that have not been delivered (and have been buffered) at the original location have to be delivered to the *MStream* at the new location. There are two design options in dealing with this problem – either forward messages from the old location to the new location or re-deliver from the sender to the new location. Forwarding messages has some negative implications for reliability and fault tolerance[3]. If the location from which the *MStream* is migrating dies before buffered messages have been forwarded to the new location, these messages will be lost. Hence, we opted for a sender-retransmission scheme. The agent daemon sending the message to an *MStream* buffers the message until it receives notification that the handler has run and the message has been consumed, re-transmitting the message on time-out. There are various tweaks that have to be incorporated to get this scheme to work right. We defer discussion of these details to a more detailed report. We are also incorporating a bulk-data transfer facility. Bulk data is transferred via a TCP connection that is established for this purpose. The transfer of bulk data is deferred until the handler is ready to execute.

Each *MStream* maintains a "time-stamp vector". There is a slot in this time-stamp vector for each target stream that the sender has ever sent a message to or received a message from. Each slot consists of four zero-initialized integers – a sent /recieved pair for the "control append" messages and a similar pair for the "data append" messages. Each *append* message originating from an *MStream* is stamped with a "control time-stamp" or a "data time-stamp" (depending on whether it is a "control" or "data" append). This time-stamp is extracted from the "sent" field of the time-stamp vector position corresponding to the *MStream* for which the *append* is targeted and this value is incremented in the time-stamp slot when the data is sent. When a message is received and consumed, the "received" time-stamp corresponding to the sender MStream in the receiver's vector is incremented so the receiver always knows which time-stamp to consume next. When an *MStream* is moved from one location to another, its time-stamp vector gets relocated to the new location. This could be an expensive operation as the size of this time-stamp is proportional to the number of slots in the vector. The cost could be reduced by transmitting only time-stamp diffs. We have not yet implemented this but plan to do so in the near term. The agent-specific state that gets moved when the system is re-configured consists of the values of the variables in each *Agent's* briefcase and any *Agent* code that does not exist at the target location. These are placed in a buffer as name-value pairs and are shipped from the source to the target location.

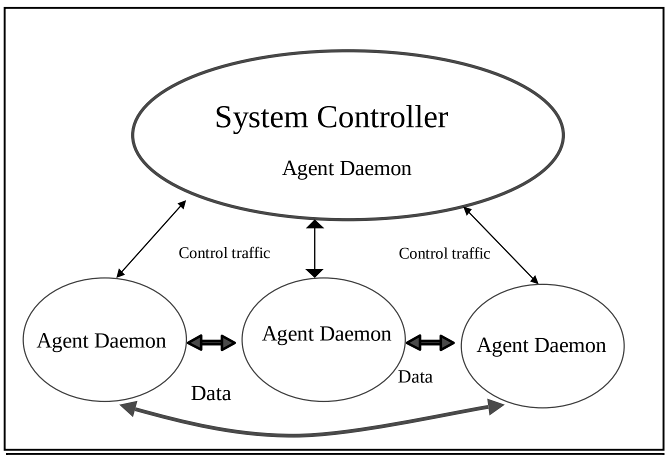

Figure 11: System Organization

Each Agent Daemon maintains a connection with a reliable "System Controller".

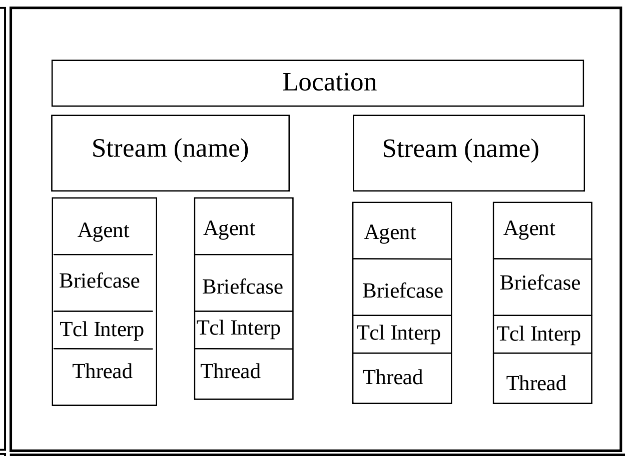

Figure 12: Logical Organization of Agent Daemon.

Each Agent its own thread and Tcl Interpreter.

The *Agent-daemons* are multi-threaded - each *Agent* is assigned a thread and each *MStream* has a separate *Stream Controller* thread[4]. On first entry of an *MStream* to an Agent Daemon, a thread and interpreter are allocated for each *Agent* associated with the *MStream*. These remain cached in the *Agent Daemon* until the time the *Agent* is destroyed (or exits). The *Agent* threads are blocked until activated by an event.

---

[3] It is our intention to eventually include fault-tolerance in our system design.
[4] We use Tcl 8.1 which is thread-safe.

We have implemented a distributed cache for the *Agent* code. When a *Stream* moves from one location to another, the *System Controller* ships the code for any *Agents* that have been added to the *MStream* and that are not present at the target location and a vector of all currently active *Agents* for the MStream. The *Agents* that are not in the "active" list are scheduled for exit processing at the target of the move. The Tcl interpreter and thread of the *Agent* are destroyed when the *Agent* exits.

6. Related Work

Our system design was motivated by considering the specific needs of distributed control, collaborative and testing applications that fit the event-driven paradigm. Our design was influenced by the design of the Meta toolkit [WM91] that also adopts the event-driven paradigm for control applications. Meta does not provide mobility and reconfiguration as our system does.

The design presented in this paper has been influenced mainly by our previous experience with designing and building the *Sumatra* system [RASS97]. Our work is also related to several other systems that offer mobility as a feature such as *TACOMA* [JVS95], *Voyager*[5] [OSVoy] , *Aglets* [V97] , *Odyssey*[6][GMOd] *ARA* [P98] and MOLE[SBH96] and to earlier systems such as such as *Emerald* [JLHB88] and *Obliq* [Ca95]. Our main innovations in this work are resource control mechanisms and separation of naming and location from functionality – features that we believe eases the task of building distributed systems. The other significant differences lie in the mobility model, the *Agent* naming scheme employed and the method of Inter-agent communication. Unlike some other systems [RASS97, G96, P98] that offer a "thread-migration model" of mobility, our system more closely resembles *Aglets, Voyager, TACOMA,* and *Odyssey*. In this section, we compare our system to three *Tcl*-based agent systems – *Agent- Tcl*, *ARA* and *TACOMA* and three *JAVA*-based systems – *Aglets* , *Odyssey* and *MOLE*.

Agent-Tcl is a Tcl-based mobile-agent system that adopts the mobile-thread model of mobility. *Agent-Tcl* adopts a very different naming and inter-*Agent* communication model than the one we have adopted. In our system, a name can exist as a communication end-point independent of *Agents*. An *Agent* is identified via the *MStream* that it is attached to. A message is sent to a collection of Agents by appending to an *MStream* rather than an individual *Agent* whereas in *Agent-Tcl,* Agents "meet" to establish a connection and send messages to each other thereafter. In our system, there is no explicit "meet" operation. Messages are sent to named communication end-points that may or may not have *Agents* attached to them. The design of *Agent-Tcl* necessitates modifying the Tcl interpreter substantially, while in our case we have not needed to do this.

ARA also adopts a "mobile-thread" model of mobility – allowing agents to migrate at arbitrary points during execution. An "*Agent-daemon*" in *ARA* is identified by a "place" with a *URL*. The *URL* also specifies the transport protocol for inter-agent transfer. The emphasis on ARA focused on inter-agent communication between Agents at a common place. ARA adopts a local message-box approach to implement this. In our system, messages are delivered to a Stream in a position-independent way. One of the focus points of our research is to investigate messaging protocols for communication between moving *Streams*. ARA allows users to specify an admission function that can determine whether or not an Agent can enter a place. The concept is similar to our Site Controller. The ARA system uses the notion of a "resource allowance" to limit the Agent's access to resources. Similarly, our system permits the System Controller to place limitations on accessible resources at a location although our approach is less general.

The *TACOMA* system is another Tcl-based *Agent* System which differs significantly in its design from our design. The *TACOMA* system adopts the view that Agents meet at a location before exchanging data. This is in contrast to our approach in which data can be sent to an *MStream* without needing to co-locate the sender and the receiver. Our notion of a briefcase is similar to *TACOMA's* notion of a briefcase – i.e. it is the data that an Agent takes with it on motion. Like our system *TACOMA* adopts the "mobile server" model. *TACOMA* has been used for building a large distributed control application called *Storm Cast* [JJSLBV96] involving distributed weather data. The system designers have identified several useful patterns or application templates as a result of having implemented this application. These templates have been implemented using the *TACOMA* system and could easily be implemented in *AGNI* in a straightforward fashion.

---

[5] *Voyager* is a Trademark of Object-space.
[6] *Odyssey* is a Trademark of General Magic Corp.

The IBM *Aglet* workbench is a *JAVA*-based mobile *Agent* system that follows a programming model that is similar to the one we have followed; however, there are several differences. First, the primary named objects in *Aglets* are *Agents*. In our design, the primary named objects are *MStreams*. A *MStream* can exist without any Agents attached to it and *Agents* may be dynamically attached to *MStreams* – thus providing a clear distinction between naming and functionality. We support grouping of code. In our design *Agents* are effectively grouped by attaching them to the same *MStream*. When the Stream moves, all Agents are moved with it[7]. Next, we support flexible grouping of data. When a Stream is moved, all the data that is in the briefcase of the Agents that are attached to it is moved. The grouping of data is dynamic - data items may be added to and removed from the briefcase dynamically. Third, movement decisions in our system can be generated from the *Agents* of the *MStream* as well as from the outside (for example from the AGNI Monitor) with the Stream Controller and Site Controller having the final say in whether the move successfully completes. Fourth, we have well-defined motion points. An *Aglet* object can receive a move command at any point and has to pack its briefcase as part of the *on_move* code. An *Agent* does not have an explicit on_move part. For *MStreams*, motion is deferred until the end of the *Handler* execution and the briefcase is made consistent at that point if the *Stream* has to move. The *Handler* boundary is a motion-cum-consistency point. This could be a disadvantage if consistency entails heavy operations such as cache flushes or disk reads. The applications we have considered so far have not needed an *on_move* handler and we may add one if the need arises.

Voyager [OSVoy] is a JAVA-based mobile agent system that resembles our system in some respects. Our model of a *Stream* with multiple *Agents* attached to the *Stream* is similar to *Voyager's* subspaces model. *Voyager* offers a federated directory service that is similar in function to AGNI's name resolution. In *Voyager*, the continuation of execution on arrival at a target location is explicitly specified as a function name. In our system, the continuation of execution is in the *on_arrival Handler*. While the *Voyager* approach provides greater flexibility, the advantage to having an explicit continuation point is again the ease of constructing debugging tools and has also facilitated the controller mechanism that we described earlier. Making the continuation points explicit also has the advantage of being able to have the system re-configure the application.

MOLE [SBH96] is another *JAVA*-based mobile-agent system that uses a "mobile-server" model of mobility. *MOLE* agents communicate with each other either by establishing a session and then exchanging messages or by global events. Our system of inter-agent communication utilizing *MStreams* comes closer to the notion of global events – with the name of the *MStream* mapping to the notion of a global event. *MOLE* implements a rule-based system for global events. Our system does not provide such a facility; however, a rule-based paradigm may be developed using our system as a base.

7. Conclusion

Our experience with *Sumatra* and *AGNI* indicates that constraining the programming model to the mobile stream model has significant benefits in terms of simplifying the implementation and improving performance[8]. This model also has the significant benefit of re-utilization of existing tools. In our system we were able to use the Tcl library unchanged whereas systems such as *Sumatra* *Agent-Tcl* and *ARA* have to modify the underlying system on which they are constructed. Going from one version of *Tcl* to another in our system (an event that occurred 3 times over the last year – not counting numerous patches) has been relatively painless and adding *Tcl* extensions developed by others is likewise painless. In contrast *Sumatra* never evolved past JDK 1.0 and we did not develop debugging tools for our system.

8. Future Work

There are several aspects of our system that we are currently refining. We are working on utilizing multicast in our system for group communication, on providing causal communication for mobile Agents and on fault-tolerance and security. We are also working on various applications including an automated test management framework for collaborative systems, a collaborative peer-to-peer caching application and network management applications.

---

[7] As described earlier, AGNI uses a distributed caching protocol to reduce the cost of this operation.

[8] At the risk of sounding pompous, our design was inspired by Einstein's words "keep things as simple as possible but no simpler".

9. Acknowledgements

This work was supported by *DARPA* funding under the Intelligent Collaboration and Visualization (IC&V) project. We are indebted to Charles Crowley for making *Tk Replay* [Cr95] freely available, to Don Libes for making the Tcl Debugger [L93] freely available and to Silvio Tribel for making Tk Draw [T98] freely available. AGNI was developed at the National Inst of Standards and Technology - Information Technology Laboratory. The NIST authors would like to thank NIST management – especially J-P Favreau, manager of the Multi-media and Digital Video group at NIST for his support and encouragement.

10. References

[ARS96] A. Acharya, M.Ranganathan and Joel Saltz, "Sumatra – A language for Resource-aware mobile programs", in *Mobile Object Systems, J. Vitek and C.Tschudin (eds), Springer Verlag Lecture Notes in Computer Science.*

[Ca95] L. Cardelli, "A Language with Distributed Scope", *proc 22nd ACM SIGPLAN-SIGACT Symposium on the principles of Programming Languages, (Jan 1995)*

[Cr95] C. Crowley, "Tk-Replay: Record and Replay in Tk", *USENIX Third Annual Tcl/Tk Workshop*, 1995.

[G96] R.S. Gray "Agent Tcl: A Flexible and Secure Mobile-Agent System", *Proceedings of the Fourth Annual Tcl/Tk Workshop (1996).*

[GMOd] General Magic Corp., http://www.genmagic.com/agents Odyssey web-page.

[JVS95] D.Johansen, R. van-Renesse, F.B.Schnieder, "An introduction to the TACOMA Distributed System", *Technical Report 95-23, Dept of Computer Science, University of Tromso, Norway, 1995.*

[JLHB88] E. Jul, H. Levy, N. Hutchinson and A. Black, "Fine-grained mobility in the Emerald System", *ACM Transactions on Computer Systems, vol 6, N2, Feb (1988).*

[JJSLBV97] D. Johansen, K Jacobsen, N.P. Sudmann, K.J. Lauvset, K.P. Birman and W. Vogels, "Using Software Design Patterns to build Distributed Environmental Monitoring Applications", *Cornell University, Department of Computer Science, TR97-1655, Dec 1997,* http://cs-tr.cs.cornell.edu/TR/CORNELLCS:TR97-1655

[L93] D. Libes, "A debugger for Tcl " *Tcl/Tk workshop, 1993.*

[RAS98] M. Ranganathan, Anurag Acharya and Joel Saltz, "Adaptive Data Combination over Wide-area netwoks", *International Conference On Distributed Computing Systems (ICDCS), 1998 (To appear).*

[RASS97] M. Ranganathan, Anurag Acharya, Shamik D.Sharma and Joel Saltz, "Network-aware Mobile Programs", *USENIX 97.*

[OSVoy] Voyager White Paper, http://www.objectspace.com/voyager

[P98] H. Peine, "Agents for Remote Access" in *Mobile Agents* by W.R.Cockayne and Michael Zyada, *Manning Publishers, ISBN 1-884777-36-8, (1998).*

[SBH96] Markus Sraβer, Joachim Baumann and Fritz Hohl, "Mole – a Java-based mobile agent system", *in 2nd. ECOOP Workshop on Mobile Object Systems. Linz, Austria, 1996.*

[T98] http://www.inf-technik.tu-ilmenau.de/~silvio/research/soft.htm Tk-Draw web page.

[V97] B. Venners, "Under the Hood: The architecture of Aglets", *Java World, April 1997.*

[WM91] M. Wood and K. Marzullo, "The Design and Implementation of Meta", in "*Reliable distributed computing with the ISIS toolkit*" by Kenneth Berman and Robert van Renesse, *IEEE Computer Society Press, ISBN 0818653425,(1994).*

Appendix

```
stream_open stream0 –controller {
     global notification_received
     set notification_received 0
     add_briefcase notification_received
     # register the control append handler – this runs when a control append
     # is consumed.
     register_control_append_handler {
         # stream_unblock unblocks the stream and runs pending appends.
        stream_unblock
        set notification_received 1
     }
     on_append_preprocessor {
        if { $notfication_received == 1 } {
           # the synchronizer has sent us the control append so we can run
           # Schedule all agents for execution.
            schedule_agents –all
           # Run all the scheduled agents.
            run_scheduled_agents
         } else {
            # block the stream and push append back into the stream
            stream_block
            # signal to the synchronizer that the append has arrived.
            stream_append synchronizer [stream_name]
         }
     }
}

stream_open synchronizer
# register an agent for the synchronizer stream.
register_agent synchronizer on_init {} {
        global status
        add_briefcase status
        set status(stream0) 0
        set status(stream1) 0
        stream_open stream0
        stream_open stream1
} on_append {
        incr status($argv)
         # got a notification from both streams
        if { $status(stream0) == $status(stream1) } {
            # make them run by sending them control appends
            stream_append stream0 "OK" -control
            stream_append stream1 "OK" -control
        }
}
```

Figure 13 Synchronizing MStreams via a Controller and Control Appends.

stream_block and stream_unblock can be used to block and unblock message processing. The control append which gets through despite the stream being blocked, unblocks the stream.