AGNI: A Multi-threaded Middleware for Distributed Scripting

M.Ranganathan, Marc Bednarek, Fernand Pors and Doug Montgomery
Internetworking Technologies Group
National Institute of Standards and Technology
100 Bureau Drive, Gaithersburg, MD 20899.
{*mranga, bednarek, pors, dougm*}*@antd.nist.gov*

Abstract

We present the design of a AGNI - a Tcl 8.1 based Middleware for building reactive, extensible, reconfigurable distributed systems, based upon an abstraction we call Mobile Streams. Using our system, a distributed, event-driven application can be scripted from a single point of control and dynamically extended and re-configured while it is in execution. Our system is suitable for building a wide variety of applications; for example, distributed test, conferencing and control-oriented applications. We illustrate the use of our system by presenting example applications.

1 Introduction

Our work is motivated by a couple of observations about the evolving nature of distributed applications and their implementation environments. First, the structure of distributed software is undergoing some changes. We are seeing the growth of new types federated, loosely coupled distributed applications that have several common requirements and characteristics including: (1)**Event-driven Architecture:** A single distributed application may be composed of separate components that all work together in a coordinated fashion. Such applications are event-oriented in nature in that we can think of changes in the overall state of the global application as being triggered by discrete changes in the state of event processing at each of the components. (2)**Heterogeneity:** The components must run on a variety of different platforms with varying inherent capabilities and environments. In addition, some of components themselves are reused pieces of software implemented in a variety of languages and environments. Despite all of this heterogeneity, it is desirable to be able to design and develop distributed applications in a common portable framework. (3)**Mobility:** The ability to dynamically move code to and among these platforms during application execution greatly enhances the ability to deploy and reconfigure complex systems. (4)**Reliability:** These extension and reconfiguration capabilities can be used to construct reliable systems that distribute system state in ways that enable graceful failure recovery and adaptation. (5)**Security:** In such highly dynamic systems, the ability to secure and control resources at several levels (global application, single node, individual process) is necessary to insure the correct behavior of applications and the viability of systems that support them.

Our second observation is that while many systems share the characteristics mentioned above, the variety of component types and platforms that must be accommodated preclude a language specific or application specific solutions. Instead we suggest that a Middleware approach based upon Tcl scripting technology provides the most flexibility in the design of such composite applications. Tcl is great "component glue". Its simplified structure and extensibility are strengths when assembling applications from disparate, heterogeneous software components [1].

This paper is about AGNI - a multi-threaded Tcl 8.1 based Middleware for scripting reconfigurable event-oriented distributed systems. AGNI builds upon the proven scripting power of Tcl by adding extensions for an abstraction we call Mobile Streams. Mobile Streams (*MStreams*) are a generalization of simple mobile code technologies (e.g. Java Applets) that provide code distribution and communication between clients and servers. MStreams extend today's simple notions of code mobility by incorporating state mobility, decentralized peer-to-peer communications and the ability to extend and reconfigure distributed application during execution while preserving behavioral guarantees. At a higher level, MStreams allow the separation of the logical structure

[1] Indeed, thus spake John Ousterhout : "If you look at financial services, a lot of what they do is try and tie together all of the different systems that need to be coordinated with traders, not to mention the front and back office. It's a tremendous integration effort, and that's exactly what Tcl does wonderfully...".

of a distributed application from the physical placement of components. Our model of code mobility allows the mapping of logical application structure to physical resources (e.g. machines and processes) to occur dynamically at run time and change during the course of the active life-time of the global application.

The rest of this paper is organized as follows: Section 2 presents the MStreams programming model and system architecture and presents an introductory example. Section 3 gives a brief overview of *AGNI*, our prototype implementation of MStreams Middleware. Section 4 presents a simulation environment for designing applications using our system. Section 5 presents some more comprehensive applications that we have built on our system. In Section 6 we compare and contrast our work with those of others. In Section 7 we conclude and present our future plans for this project.

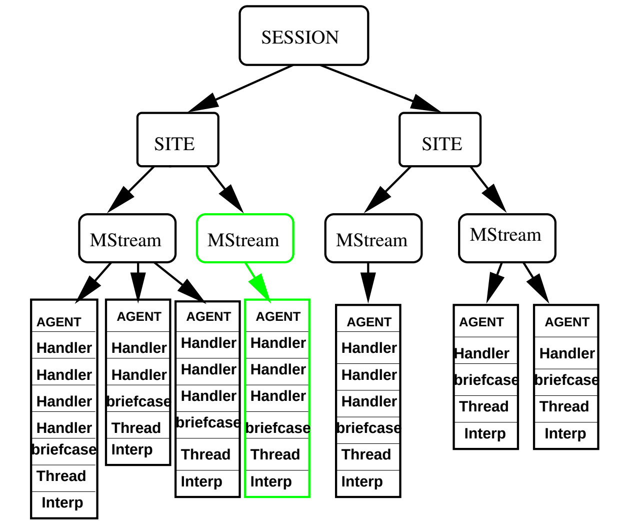


Figure 1: Logical organization of the System. A Session consists of multiple participating Sites. Each Site can house multiple MStreams. Each MStream can have multiple Agents that can register Handlers for different Events. MStreams can move from Site to Site. When an MStream moves, all its registered handlers move with it.

2 Mobile Streams

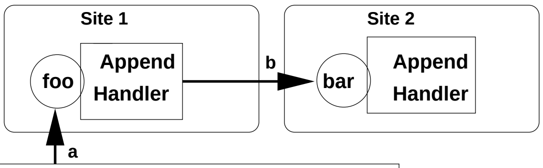
In this section we present our programming model and provide a small, introductory example. We begin by introducing a few terms that are used through the rest of the paper.

A *Mobile Stream* (*MStream*) is a mobile communication endpoint in a distributed system. The closest analogy to an MStream is a mobile active mailbox. As in a mailbox, an MStream has a globally unique name. *MStreams* provide a *FIFO* ordering guarantee, ensuring that messages are consumed at the MStream in the same order as they are sent to it. Usually mailboxes are stationary. MStreams, on the other hand, have the ability to move from *Site* to *Site* dynamically. Usually mailboxes are passive. In contrast, message arrival at an MStream potentially triggers the concurrent execution of message consumption event handlers (*Append Handlers*) registered with the MStream, which can process the message and, in turn, send (*append*) messages to other MStreams.

An MStream has a globally unique name. We refer to any processor that supports an MStream execution environment as a *Site*. A distributed system consists of one or more Sites. A collection of Sites participating a distributed application is called a *Session*. Each Session has a distinguished, trusted, reliable Site called a *Session Leader*. Each Site is assigned a *Location Identifier* that uniquely identifies it within a given Session. New Sites may be added and removed from the Session at any time. An MStream may be located on, or moved to any Site in the Session that allows it to reside there. MStreams may be opened like sockets and messages sent (appended) to them. Multiple Event Handlers (*Handlers*) may be dynamically attached, to and detached from, an MStream. Handlers are invoked on discrete changes in system state such as message delivery (append), MStream relocations, new Handler attachments new Site additions and Site failures. We refer to these discrete changes in system state as Events. Handlers are attached by *Agents* which provide an execution environment and thread for the Handlers that they attach. (i.e. an Agent specifies a collection of Handlers that that all use the same thread of execution and interpreter.) Logically, the system is structured as shown in Figure 1.

Handlers can communicate with each other by appending messages to MStreams. These messages are delivered asynchronously to the registered Append Handlers in the same order that they were issued [2]. A message is *delivered* at an MStream when the Append Handlers of the MStream has been activated for execution as a result of the message. A message is *consumed* when all the Append Handlers of the MStream that are activated as a result of its delivery have completed execution. By *asynchronous delivery* we mean that the sender does not block until the message has been consumed in order to continue its execution.

[2]Synchronous delivery of messages is supported as an option but asynchronous delivery is expected to be the common case.




```
External Input
stream_open foo
stream_append foo"Hello world" ;#a
```

```
stream_create foo
stream_create bar
register_agent foo {} {
    stream_open bar
    on_stream_append {
        stream_append bar $argv ;#b
    }
}
register_agent bar {} {
    on_stream_append {
        puts $argv
        stream_relocate 1 ;#c
    }
    on_stream_relocation {
        set my_loc [stream_location]
        puts "I am at $my_loc" ;#d
    }
}
stream_move foo 1
stream_move bar 2
```

Figure 2: A simple auto-reconfiguring reactive system scripted from a single point of control.

A distributed application is constructed by first specifying the communication end-points as MStreams and then attaching Agents to those end-points, that in turn attach Handlers for specific Events. A given MStream may have multiple Agents and each Agent may register Handlers for different Events, but each Agent may have only one Handler for a given Event. When an Event occurs, the appropriate Handlers in each Agent are concurrently and independently invoked with appropriate arguments. Handlers are typically registered on Agent initialization and may be dynamically changed during execution.

An application built using our Middleware, may be thought of as consisting of two distinct parts - an active part and a reactive part. The reactive part consists of MStreams and Handlers. The active part or *Shell* lives outside the Middleware and drives it. A Shell may connect to the Middleware and issue requests and may exit at any time. The reactive part is persistent.

Figure 2 shows an example script that instantiates a simple distributed system that resides at *Sites* 1 and 2. A message is sent to the MStream called *foo* by the `stream_append` command issued via the external Shell (#a in Figure 2) . The MStream called *foo* receives the message "Hello world" and sends it to the MStream called *bar* (#b in Figure 2) which outputs the message via its handler and then moves MStream *bar* to Site 1 (#c in Figure 2). The arrival handlers run when the MStream *bar* arrives at *Site* 1, printing the string "I am at 1" to the console at Site 1 (#d in Figure 2).

In Figure 2 the script labelled "External Input" in is the Shell and MStreams and their registered handlers are the reactive parts.

2.1 Dynamic Extension and Re-configuration

An application built on our Middleware may be dynamically extended and re-configured in several ways while it is in execution (i.e., while there are pending un-delivered messages). First, an Agent can dynamically change the handlers it has registered for a given Event. Second, new Agents may be added and existing Agents removed for an existing MStream. Third, new MStreams may be added and removed. Fourth, new Sites may be added and removed, and finally, MStreams may be moved dynamically from Site to Site.

When an MStream moves from one Site to another, it (logically) moves the code of all of the Agents attached to it to the new Site along with whatever state they have placed in their *briefcase* structures. We say an Agent "visits" a Site when its MStream visits the Site. When an Agent first visits a Site, its initialization code executes there and when an Agent is killed, its (optional) *Finalization Handler* runs at each location that has been visited by it. Agent state (consisting of global state variables and code) is replicated at each site that it visits until the Agent is destroyed. On Agent destruction, the Handlers that it has registered are de-registered, and the interpreter and state variables are freed at each Site that it has visited. We assume that Sites may fail or disconnect during execution. Site failure does not imply destruction of the MStreams that reside there. Failure processing is described in Section 2.3.

The Agent's briefcase specifies a consistency requirement for moves. When an Agent moves from Site to Site only the elements in the briefcase are copied from the source execution environment to the target. The remainder of the global state remains unaffected (and cached) at the source site of the move. On successful completion of a move, the *Arrival Handlers* of the MStream are invoked at the new Site where the MStream has moved.

Handlers may move the MStream to which they are attached and also may move other MStreams around as well as create and destroy MStreams. Handlers may also exit - destroying the Agent in which they are housed and may also destroy other Agents. Such actions may also be initiated from an external Shell. Re-configuration may be contained by using appropriate policy handlers.

All changes in the configuration of an MStream such as MStream movement, new Agent addition and deletion, and MStream destruction are deferred until the time when no Handlers of the MStream are executing. We call this the *Atomic Handler Execution Model* . Message delivery order is preserved despite dynamic reconfiguration, allowing both the sender and receiver to be in motion while asynchronous messages are pending delivery.

Applications built using Mobile Streams can be extended from multiple points of control; any handler or Shell that has acquired an open MStream handle, can attempt to re-configure or extend the reactive part of the system and these actions can occur concurrently. While this adds great flexibility, it also raises several security and stability issues. We provide a means of restricting system reconfiguration and extension using control Events that can invoke policy Handlers. These policy Handlers may be registered only by privileged Agents as described below. We follow a discretionary control philosophy by providing just the mechanism and leaving the policy up to individual applications. Controls may be placed via policy Handlers at a session-wide level, site-wide level and at the level of individual MStreams for various security-relevant Events.

In summary, our security mechanisms are based on the following three principles:

Session-wide control: We have built mechanisms to place session-wide controls over extension and reconfiguration via a centralized *Session Leader* MStream.

Site-specific control: Each site may specify security policies via a *Site Controller* MStream. Site-specific policies may be used to grant or deny MStream entry to a site and to sand-box the incoming MStream handler's code by using safe-Tcl mechanisms.

Stream-specific control: The MStream itself is regarded as an extensible entity to which Agents can be attached and detached. It can carry its own policy Handlers to allow or disallow such actions, as determined by its *Stream Controller* Agent.

Our security implementation works as follows: Messages are classified as data messages and control messages. Data messages are delivered directly to the MStream. Control messages are messages that can change the configuration of the distributed system. These are routed first through the trusted intermediary Session Leader that can accept or deny these actions via its policy handlers, and then through the Site Controller which may again accept or deny the action and finally through the Stream Controller for MStream-specific actions. We invite the interested reader to look at [9] for more details.

The mechanisms described above permit us to build highly flexible and extensible distributed reactive systems that are able to extend and re-configure themselves, and also to place constraints on how the system can be re-configured and extended.

2.2 Message Delivery

Within our Middleware framework, point-to-point messages are delivered using an in-order sender-reliable delivery scheme built on top of UDP. All messages are consumed in the order they are issued by the sender despite failures and reconfigurations. These ordering and delivery guarantees make it simpler to design distributed systems.

In our scheme, the sender of the message is responsible for re-transmitting the message on timeout. We use a sliding-window acknowledgement mechanism similar to those employed by TCP. The sending Site buffers the message and computes a smoothed estimate of the expected round-trip time for the acknowledgment to arrive from the receiver. If the acknowledgment does not arrive in the expected time, the sender re-transmits the message. The sender keeps a window of unacknowledged messages and controls flow by dynamically adjusting the width of this window depending upon whether an ACK was received in the expected time or not. Thus far, our description is similar to the mechanisms employed by TCP. We have implemented our own protocol, rather than just use TCP, because TCP does not address certain conditions such as failures above the transport level and dynamic movement of the communicating end-points.

As previously described, an application can be dynamically reconfigured at any time with both the sender and receiver moving. When movement of an MStream occurs, a *Location Manager* is informed of the new Site location where the MStream will reside. This information needs to be propagated to each Handler or Shell that has opened the MStream.

When the target of an Append moves, messages that have not been consumed have to be delivered to the MStream at the new Site. There are two design options in dealing with this problem - either forward un-consumed messages from the old Site to the new Site or re-deliver from the sender to the new Site. Forwarding messages has some negative implications for reliability. If the Site from which the MStream is migrating dies before buffered messages have been forwarded to the new Site, these messages will be lost. Hence, we opted for a sender-initiated retransmission scheme. The sender 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.

When an MStream moves it takes various state information along with it. Clearly, there is an implicit movement of handler code and Agent execution state (via the briefcase), but in addition, the MStream takes a state vector of sequence numbers. There is a slot in this vector for each "alive" MStream that the MStream in motion has sent messages to or received messages from. Each slot contains a sent-received pair of integers indicating the next sequence number to be sent or received from a given MStream. This allows the messaging code to determine how to stamp the next outgoing message or what sequence number should be consumed next from a given sending MStream.

2.3 Handling Failures

A failure occurs when the Site where the MStream resides fails or disconnects from the Session Leader. Each MStream is assigned a reliable *Failure Manager* Site. When a such a failure occurs each of the MStreams located at the Site that has failed are implicitly relocated to its Failure Manager Site where its Failure Handlers are invoked. Failures may occur and be handled at any time - including during system configuration and reconfiguration. Pending messages are delivered in order, despite failures. A message is considered "consumed" only after all of the Append handlers execute at the target MStream for that message. (If none exist the message is discarded at the recipient). If the Site housing an MStream should fail or disconnect while a message is being consumed or while there are messages that have been buffered and not yet delivered, re-delivery is attempted at the MStream Failure Manager. To ensure in-order delivery in the presence of failures, the message is discarded at the sender only after the Append Handlers at the receiver have completed execution and the ACK for the message has been received by the sender. This is different from TCP where the receiver ACKs the message immediately after reception (and not after consumption as we require). After a failure has occurred at the site where an MStream resides, a failure recovery protocol is executed that re-synchronizes sequence numbers between communicating MStreams that involve the failed MStream. Each of the potential senders is queried to obtain the next expected sequence number. FIFO ordering can be thus be preserved despite the failure.

3 Implementation

We have implemented the Mobile Streams model in a toolkit we call AGNI [3] . AGNI is a multi-threaded TCL extension that uses the thread-safety features of TCL 8.1 and consists of roughly 23,000 lines of C++ code. In this section, we give highlights of the implementation some initial performance results.

Each workstation that wishes to participate in the distributed system runs a copy of an *Agent Daemon*. A distinguished Agent Daemon houses the Session Leader and is in charge of accepting or rejecting new Agent Daemons. This Daemon also serves as a Location Manager and Failure Manager for all MStreams in the Session. Each Agent Daemon has a unique identifier that it obtains from the Session Leader. Each Agent Daemon maintains a connection with the Session Leader Agent Daemon. Conceptually, the arrangement is as shown in the figure 3.

Each Agent has a TCL interpreter and thread of execution that is used by the Handlers that it registers. These resources are created for an Agent at a Site on its the first visit to the Site and remains allocated until the Agent (or the MStream to which it is attached) is destroyed. When a new Agent is added to an MStream, its code is propagated and initialized on the first move of the Agent to a previously unvisited Site, and remains cached there until it is destroyed. Provided an MStream has visited a Site previously, and no new Agents have been attached since its last visit, MStream movement simply consists of moving the state information in the briefcase (see Section 2) of each Agent of the MStream to the new Site and concurrently invoking each *on_stream_arrival* Handler.

Except for the case when the MStream is co-located with the Site from where the message originates, all control Events destined for an MStream (e.g. creation, relocation, new agent attachment) are delivered through the Session Leader Agent Daemon via the TCP connection that

[3] "AGents at NIst" (also Sanskrit for fire)

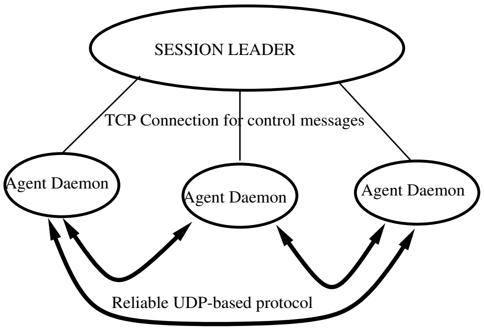


Figure 3: Each Site runs an Agent Daemon that is connected to the Session Leader. The Agent Daemon is Multi-threaded with one thread per agent. The Session Leader maintains location and cache information.

each Agent Daemon maintains with it. The Session Leader Agent Daemon also acts as a Location Manager, keeping track of where each MStream is located and is hence able to re-direct messages to the location of the MStream. Sending all control Events through the Session Leader is a simple means of achieving a global ordering on control messages. The negative aspect of this design is that the Session Leader has the potential of becoming a bottleneck. However, we expect the number of control messages to be much smaller than the number of data messages (appends) processed by the MStream and hence do not consider this a serious limitation at present. In our future work, we plan to alleviate this problem by replication of the Session Leader. The Session Leader Daemon also manages the tracking information for the code and state cache described previously and is charge of propagating code to previously unvisited locations. As all Agent code is registered at the Session Leader and propagated from there, this simplifies the trust model to pair-wise relationships between each site and the Session Leader. provided all parties trust the Session Leader.

Appended data messages are delivered to the destination MStream directly without going through the Session Leader. Thus the Session Leader is not a bottleneck for data message delivery.

4 An In-Situ Simulation Environment

Estimating the detailed behavior and performance of a distributed system is hard. There are several degrees of variability and the interaction between physical effects is often difficult determine. Further, bugs - especially timing related ones can be quite difficult to reproduce in physical test-bed environments. In order to address these issues, we have developed an in-situ simulation environment that enables tuning, debugging and performance estimation of both the AGNI runtime system and applications.

Our approach system wraps a simulated environment around the actual AGNI system and application code using the CSIM [11] simulation library. We replace thread creations, locking and message sends and receives with simulated versions of these but leave the rest of the code unmodified. We have used the simulation for debugging and performance tuning the system as well as for testing the performance of applications built on top of our system. Figure 4 shows the simulation code for the introductory application presented in section 2. As can be seen from the example, the simulation and the actual system script look quite similar, with the exception of the parameters at the top and some new commands to create simulated processes and shells. The simulation runs as a single process whereas the actual system consists of multiple communicating processes. The simulation contains various ”tweaking” parameters that have to be adjusted to match reality. These parameters include the message latency, packet drop percentage and simulated delays corresponding to code execution time. The goal in tuning the simulation is to adjust these parameters to make the simulation match the behavior of the real system for the quantities of interest. Presumably, we can match these over some simple scenarios and then try more complex ones, having some assurance of the validity of results. One can get a good idea about what delays are significant by looking at the gprof execution trace for the actual system.

A simulation is, however, only good to the extent it matches reality. Fitting the simulation to reality involves several cycles of adjusting performance parameters and re-testing the simulation. There is a large degree of variability in the performance of the actual system. We aim to make the output of the model fall within one standard deviation of the actual system for the quantities of interest. To test if this is feasible, we tested some simple scenarios. In both the real and simulated environments, we emulated packet drop by randomly dropping packets at the receiver. The quantities of interest that we would like to match are the throughput of packets and the number of packets sent by the sender for each packet consumed by the receiver (packet ratio). While we are still in the process of tuning the simulation, our initial results are encouraging. Figure 5 shows the message count performance of the real system and simulated system for two fixed end-points.

```
set drop  10
set m0 [ machine m0 ] ;#1
set m1 [ machine m1 ]
set e0 [ create_engine $m0 -d $drop ] ;#2
set e1 [ create_engine $m1 -d $drop ]
sim_set_send_latency m0  .0005 ;#3
sim_set_send_latency m1  .0005
sim_set_execution_time "Tcl_FindHashEntry" 0.000015 ;#4
sim_set_execution_time "Tcl_NextHashEntry" 0.000015
sim_set_execution_time "Arrival" 0.000015       ;#5
create_shell $e0 {
  stream_create foo
  stream_create bar
  register_agent foo {} {
        stream_open bar
        on_stream_append {
            stream_append bar $argv
        }

  }
  register_agent bar {} {
        on_stream_append {
                puts "$argv"
                stream_relocate 1
        }
        on_stream_relocation {
             puts "I am at [stream_location] at [sim_clock]"
             conclude_sim
        }
  }
  stream_move foo 0
  stream_move bar 1
  stream_append foo "bar"
}
sim 100000
```

Figure 4: Simulation script for introductory example.

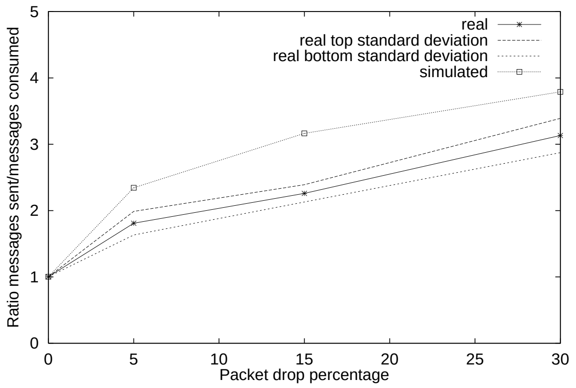


Figure 5: Simulated versus actual packet ratio for fixed end-points.

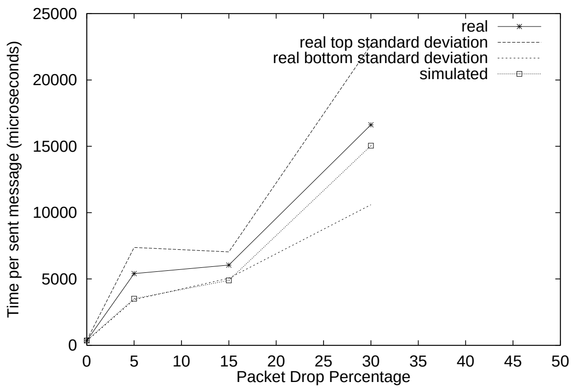


Figure 6: Simulated versus actual message consumption time for fixed end-points.

5 Application Sketches

In building applications using our infrastructure we adopted a problem-driven approach in mapping AGNI capabilities to prototype solutions. For example, we started with the assumption that we would use mobility only to the extent that it simplified the application design or enhanced performance in some fashion, rather than adopt the approach that mobility is a feature whose utility needed to be demonstrated. The remainder of this section outlines the design of two applications. The interested reader is referred to [9] for additional examples.

5.1 Distributed Data Combination

Consider a distributed experiment where data is being gathered at multiple sites and a query involves picking up data items from each location and combining the data to produce a composite result. Such an application may be structured as a master server front end and a multiple slave back ends. The front end gets the query and farms it out to to each of the participating sites. The slave sites process sub-queries locally, gathering results and returning them to the master site which then returns the combined result to the remote caller. If the combination operation is an involved one such as a database join, this could place an excessive burden on the master. Alternatively, this operation could be offloaded to the client or one of the slaves. When the query is received by the master, it creates an MStream at the client or one of the slaves to receive data from the data sources and and process the join.

If the data can be shipped incrementally from the data source, and results can be produced incrementally, the operation that receives results can be moved dynamically between the slaves and the client depending on available bandwidth and other machine resources. Such techniques are useful for optimizing dynamic query execution in client-server database systems. The ordering guarantee provided by MStreams ensures that the incremental join results are received in order by the join operator and the output operator regardless of the physical location of the join operator. In particular the join operator could be dynamically moving between sites as the join is being processed. Several positioning strategies may be considered in dynamically moving the join operator around. Some of these strategies are considered in [7] where we consider the more complex case of a join tree and adapt the operator placement to bandwidth variations. Figure 7 shows the overall organization of the system.

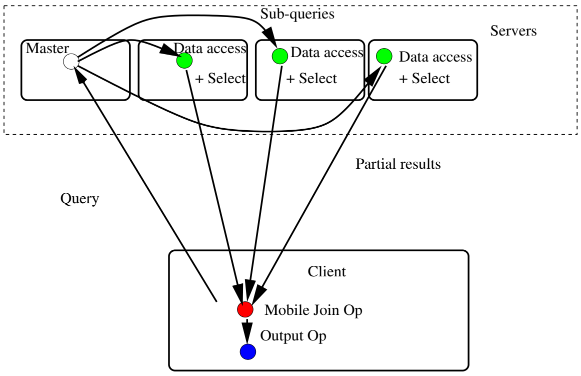


Figure 7: Adaptive database query execution. The join operator may be dynamically repositioned while the join is in progress.

5.2 Collaborative Annotation of Experimental Data

Although we have described our tool primarily as a means of scripting distributed applications, there is no requirement that the MStreams reside on physically separated machines. In section we describe SMAT - A Synchronous Multimedia Annotation Tool. SMAT was designed to be part of a scientific collaboratory for use in a robotic arc welding research project at NIST [13].

The scenario is as follows: Data is produced by sensors in various parts of a welding system and welding cell controller. Data from these sensors can have different media types - for example, video, audio and discretely sampled current and voltage. The primary functional requirement for SMAT was to develop a tool that supports the capability synchronize and play back the captured multi-media data after the weld is complete and provides a means to stop the playback at any point in time and enter annotations. After the annotation session is complete, the entered annotations are uploaded to an server for other users to view and annotate. During subsequent sessions, the media and the annotations are played back in synchronous fashion. Annotations appear in the annotation window corresponding to the relative time at which they were entered.

A secondary requirement for SMAT was to support real-time collaboration in the tool. Using this capability, users may effectively have partial control over each other's tools in order to share the same view of the multimedia data.

To meet these requirements, SMAT was designed as a control and integration framework that exploits existing tools to play specific media types. We started with the assumption that each tool to be controlled exports an API or mechanism (such as COM) that permits it to be controlled from another process. The tools are all tied together using a common *control bus* implemented using MStreams. The idea of the bus is much the same as the idea of a bus in computer hardware. Components are tied together by plugging them into the software bus in the same fashion as cards are plugged into a hardware bus. The components in this case are slave processes that play the different multimedia files. In order to use such an approach, the interfaces to the tools under control must be made uniform. To achieve this uniformity we wrap a controller script around each tool. For example, we can use XANIM as a tool that plays video under UNIX. XANIM takes external input via property change notifications on an XWindow Property. If we use a Microsoft tool, it may export COM interfaces for external control. In general, each tool may have its own idiosyncrasies for external communication. We encapsulate these via a software driver wrapper that hides the communication complexities from the control layer and registers standardized callbacks with the control layer. This is modeled after a device driver in an operating system that would register *read*, *write*, *ioctl*, *open* and *close* callbacks. The callbacks in our case include a *start* interface, a *stop* interface, a *quit* interface, a *timer tick* interface and a *seek* interface. These get called from the controller at appropriate times. It is up to the driver to communicate with the slave tool if need be on each of these calls. To enhance usability, we need the look and feel of a single tool rather than several individual tools. For this, we use Tk window embedding. Each tool that has a embedable top level window is embedded in a common canvas. The overall tool is controlled by the user via a control GUI that also sends events through the control bus. The architecture is shown in Figure 8.

There are several advantages to structuring a tool in this fashion:

Distributed Control Each tool is controlled by a separate AGNI Agent that implements its driver. The driver reacts to events that can be generated from anywhere in the distributed application. For example, the slider tool can append messages to the controller that re-distributes these events as *seek* events to each of the tool drivers. If the multimedia tools support random seeks, they can respond to such seek requests and position their media appropriately, thereby giving the ability to have both real-time and manually controlled synchronization. If we wanted to share the slider, in a synchronously collaborative fashion, this seek input simply needs to originate from another machine rather than the local slider. The control inputs could also come from another collaborative environment and in-

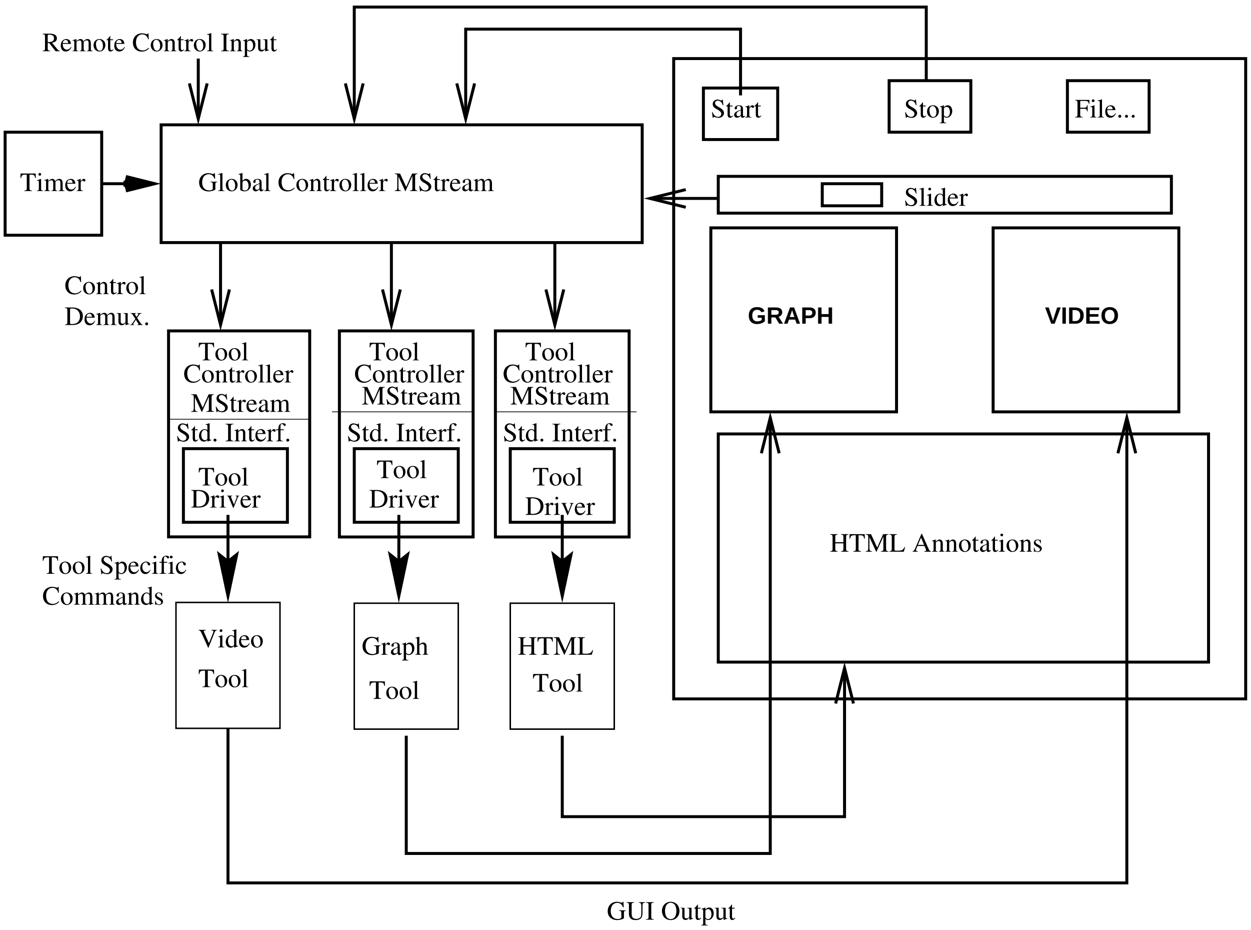


Figure 8: SMAT: A composite annotation tool with a distributed control bus. Each media type is handled by a separate tool with its own driver. An MStream-based event-bus is used to tie together the tools and provide a means to selectively export controls.

deed we have used this approach to integrate the tool in with the Teamwave client [12].

Isolation of Components Each tool runs in its own address space. Thus, a misbehaving tool cannot bring down the application. Failures are easy to isolate and fix. We can utilize off-the-shelf tools for media handling and annotation whenever such tools are available. For example, in our Windows NT version of the tool, we use the COM *IWebBrowser2* interfaces to Windows Explorer and drive it as an external tool to allow us to browse annotations. We use the COM *IDispatch* interface to Microsoft Word to bring up an editor to enter annotations.

Modularity and Extensibility: As all drivers export uniform interfaces, it is easy to add new media types. We simply build a driver to encapsulate the interface to the tool and plug it into the bus.

A practical issue that arises in this design is how to deal with cleanup. When the main interface exits or is killed the entire tool including all its components should be terminated. To deal with this problem, we use the *client_attach* and *client_detach* events for which the Site Controller MStream may register Handlers. These handlers are executed when a client attaches or detaches from the daemon at a given site. It can issue messages to the other tool controllers MStreams to exit the tools that they control.

It may be a concern that the decomposition of the system into processes degrades performance. Our experience was that degradation in performance is not unacceptable. The system appeared to behave well even on a slow machine (130 MHz) running windows NT.

We are also working on a data collection facility that will monitor the system, gather data and populate ftp repositories with the data after experiments are completed.

6 Related Work

Tcl DP [10] is the most popular extension for distributed scripting. Our first point of comparison is with this system. In contrast to Tcl DP that is RPC oriented, our system is intended as a platform to script distributed event-oriented applications. We rely on one-way messages to support this. In Tcl DP, messages are round-trip and the sender cannot proceed until the recipient has completed processing. Our system can also support synchronous (round-trip) messages where the sender blocks until the append handler at the target completes execution and hence we can do the kinds of things Tcl DP is aimed at doing. However, we expect most applications built using our system to be one-way message oriented. It is interesting to note that in our system, we can move the server in response to an RPC before the reply comes back to the client.

Our framework and toolkit is related to several other systems that support mobility. In contrast to other research in Mobile Agents, our approach has been to treat mobility and Mobile Agent technology as an enhancement to distributed scripting rather than as a means of supporting disconnected operations. Consequently, we have concentrated on typical distributed systems issues such as location tracking, message passing and failure handling. This distinguishes and separates our work from the other work in this area. Tcl provides an ideal platform for building mobile agent systems and there have been a few such systems that have gained popularity. Agent Tcl [3] supports a generalized mobility model where migration is allowed at arbitrary points in execution of the mobile code. This provides greater flexibility and perhaps a more natural programming model than we provide. However, this approach suffers from a few shortcomings. First, it requires modification of the core Tcl distribution - something that is difficult to keep up with over the long run. Unrestricted mobility makes support of fault tolerance and reconfiguration harder to achieve. In contrast, our system restricts mobility and other state changes to handler boundaries and treats handlers as atomic units of execution. By providing such a clean execution model, we simplify the system design and implementation while increasing slightly the burden of the developer using our system. Previously, we had developed a system called *Sumatra* that supports unrestricted mobility for Java applications by modification of the Java Virtual Machine [8] and many of the design decisions in this system are influenced by the experience gained in the Sumatra exercise. TACOMA [5] is another Tcl-based mobile agent system that adopts a programming model similar to ours. However, there are some basic difference as outlined below.

In this work, we proposed direct communication (reliable message passing) between Mobile Agents. In our system *on_stream_append* Handlers (analogous to "Agents" in other systems) pass one-way messages to each other reliably (via MStreams) rather than meeting to exchange messages, using a blackboard or other RPC-like mechanisms. Cabri et. al. [1] argue that this is not such a good idea for several reasons which make sense in the context of free-roaming disconnected agents. Our system is oriented towards building re-configurable distributed applications rather than supporting free-roaming autonomous entities and hence several of their concerns do not apply.

Aglets [6], Voyager [2], and Mole [14] are Java-based systems that follow a programming model similar to ours. However, our system differs from these systems in the following important ways: (1) Our design philosophy is to incorporate reconfiguration into a distributed system building toolkit rather than support disconnected operation as the fundamental design goal, (2) We have incorporated a peer-to-peer reliable, resilient message delivery protocol that none of these other systems offer and (3) We have a means of restricting system re-configuration and extension using policy Handlers that separate global (system-wide), and local concerns.

Dynamic re-configuration of distributed systems has been considered by Hofmeister and Purtilo [4] using a software bus approach. Their system supports dynamic changes to modules, geometry and structure of a distributed system. However, failure processing and asynchronous message delivery during reconfiguration is not considered.

7 Conclusions and Future Work

In this paper we have presented the motivation and design of a Middleware framework that uses mobility to simplify distributed scripting. We presented examples to illustrate the use of our system. Our system may be downloaded from from http://www.antd.nist.gov/itg/agni/.

Our plans for extending the Middleware is concentrated in three areas. We will incorporate reliable multicast primitives in our system whereby an MStream can communicate with a group of MStreams. As in the unicast case, both the sender and the recipients can be in motion while messages are being delivered. Second, we intend to make our location tracking scheme more robust and scalable by using replication and multicast. Third, we will build persistence at the location manager so that the system can be stopped and restarted without loosing all the MStreams and data. Finally, we intend to continue building applications - especially in the domain of mobile computing and distributed testing.

8 Acknowledgments

The authors acknowledge and appreciate the contributions of Virginie Schaal (NIST), Virginie Galtier (NIST), Laurent Andrey (LORIA, Fr.) and Anurag Acharya (UCSB) to this project. We thank Kevin Mills, Mark Carson and Craig Hunt of NIST, for reading this paper and making useful suggestions to improve its content, readability and presentation.

References

[1] G. Cabri, L. Leornardi, and F. Zambonelli. Coordination in mobile agent systems. Technical Report DSI-97-24, Universita' di Modena, October 1997.

[2] Object Space Corp. Voyager white paper. http://www.objectspace.com/voyager.

[3] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop - Monterey CA*, July 1996. http://www.cs.dartmouth.edu/ agent/papers.html.

[4] Christine R. Hofmeister and James M. Purtilo. Dynamic reconfiguration of distributed programs. In *11th. International Conference on Distributed Computing Systems*, pages 560–571, 1991.

[5] Dag Johansen, Robbert van Renesse, and Fred B.Schnieder. An introduction to the TACOMA distributed system. Technical Report 95-23, University of Tromso, Norway, June 1995. *http://www.cs.uit.no/DOS/Tacoma/tacoma.webpages*.

[6] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998. ISBN 0-201-32582-9.

[7] M.Ranganathan, Anurag Acharya, and Joel Saltz. Adapting to bandwidth variations in wide-area data access. In *International Conference On Distributed Computing Systems*, pages 498–506, May 1998.

[8] M. Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware mobile programs. In *USENIX Winter Technical Conference*, jan 1997.

[9] M. Ranganathan, V. Schaal, V. Galtier, and D. Montgomery. Mobile streams: A middleware for reconfigurable distributed scripting. In *Agent Systems And Architectures/Mobile Agents '99 (to appear)*, October 1999.

[10] Brian Smith, Tibor Janosi, and Mike Perham. Tcl dp. http://www.cs.cornell.edu/Info/Projects/zeno/Projects/Tcl-DP.html.

[11] Mesquite Software. Csim-18 simulation library. http://www.mesquite.com.

[12] Teamwave Software. Teamwave collaborative toolkit. http://www.teamwave.com.

[13] Michelle Steves, Wo Chang, and Amy Knutilla. Supporting Manufacturing Process Analysis and Trouble Shooting with ACTS. In *IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)* , June 1999. http://www.mel.nist.gov/msidstaff/steves.micky.html.

[14] Markus Straer, Joachim Baumann, and Fritz Hohl. Mole – a Java based mobile agent system. In *2nd ECOOP Workshop on Mobile Object Systems*, pages 28–35, Linz, Austria, July 1996. http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/1996-strasser-01.ps.gz.