

# A Toolkit for Re-configurable Distributed Scripting

M.Ranganathan , Laurent Andrey, Virginie Schaal and Jean-Philippe Favreau  
{mranga,andrey,schaal,favreau}@snad.ncsl.nist.gov

Multimedia and Digital Video Group  
National Institute of Standards and Technology  
Gaithersburg  
MD 20899

## *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 allows the construction of distributed applications that can be dynamically reconfigured and extended while in execution. This feature can be exploited by constructing dynamic applications that can adapt to available resources. In this paper we describe our toolkit and an example of its use.*

## **1. Introduction**

A large class of distributed collaborative, testing, monitoring and control applications fit the “event-driven” paradigm. An event-driven application is driven by asynchronous external inputs that cause event-handlers to be invoked. In a synchronous collaborative system, events are triggered by user inputs. In an asynchronous collaborative system, events may be triggered by arrival of mail or other notification from other participants. In a distributed monitoring and control system, events are triggered by transducer inputs. In a distributed testing scenario, events are triggered by test outputs, timer timeouts and so on.

We have developed an application building framework called AGNI<sup>1</sup> that facilitates the development of event-driven distributed applications. Our system is dynamically extensible - we have the ability to add and remove active components while the system is in execution. Our system is also dynamically re-configurable – active components can be relocated at well-defined points in the computation while the system is in execution. These features are useful in the design of distributed, event-oriented systems such as collaborative systems where computational resources, user requirements and loads are not known a-priori.

In this paper we describe the design of our system and an example application.

## **2. Overview of the Programming Model**

The important abstractions in our system are *Locations*, *Streams*, *Agents* and *Events*. We refer to a communication end-point as a *Stream*. A *Stream* has a globally unique name. Messages may be sent to streams and are consumed in a well-defined order. A *Stream* may reside on any workstation in the distributed system that hosts an execution environment for it. We call such an execution environment an *Agent Daemon* and refer to it with a unique identifier called a *Location*. At any given time, a *Stream* may reside in only *one Agent-daemon* (i.e. it has a unique Location referred to as the “Location of the Stream”). When a message arrives at a *Stream*, it may trigger the execution of the *Agents* associated with the *Stream*. *Agents* can be dynamically attached to and removed from *Streams* and the location of the *Stream* may be decided dynamically while the system is in execution. *Streams* may also be created on the fly.

---

<sup>1</sup> Agents at NIST (also Sanskrit for “fire”)

For a *Stream*, we define two “events”. An *Agent* may associate a code fragment with each of these events. The event-specific code fragment of each of the *Agents* associated with a *Stream*, is executed concurrently when the event occurs. The two Stream-specific events are: (1) The *on-append* event that occurs when a message is consumed at a Stream. At message consumption, each of the on-append scripts is executed concurrently. (2) The *on-relocation* event occurs at the target location when a *Stream* is re-located from one *Agent-daemon* to another.

In addition, there are two *Agent-specific* “events”: (1) the *on-init* event which occurs when an agent is first initialized at a given location. As the stream moves from location to location, the *on-init* part of the *Agent* is executed exactly once for each new location that it visits. (2) The *on-exit* event occurs when the agent is killed or exits. The purpose of this event is resource cleanup. The *on-exit* script associated with this event runs at each location where the agent has been initialized but its execution may be deferred until the next *Stream* visit to the location. An *Agent* may specify a portion of global state as being in its *briefcase*. This state is re-located (and instantiated in the environment of the *Agent* at the target location) when the system is reconfigured.

## 2.1 Resource Control

Any component of an application, external or internal, can initiate re-configuration or extension of the system. The purpose of the resource-control mechanism is to provide a means to restrict this power.

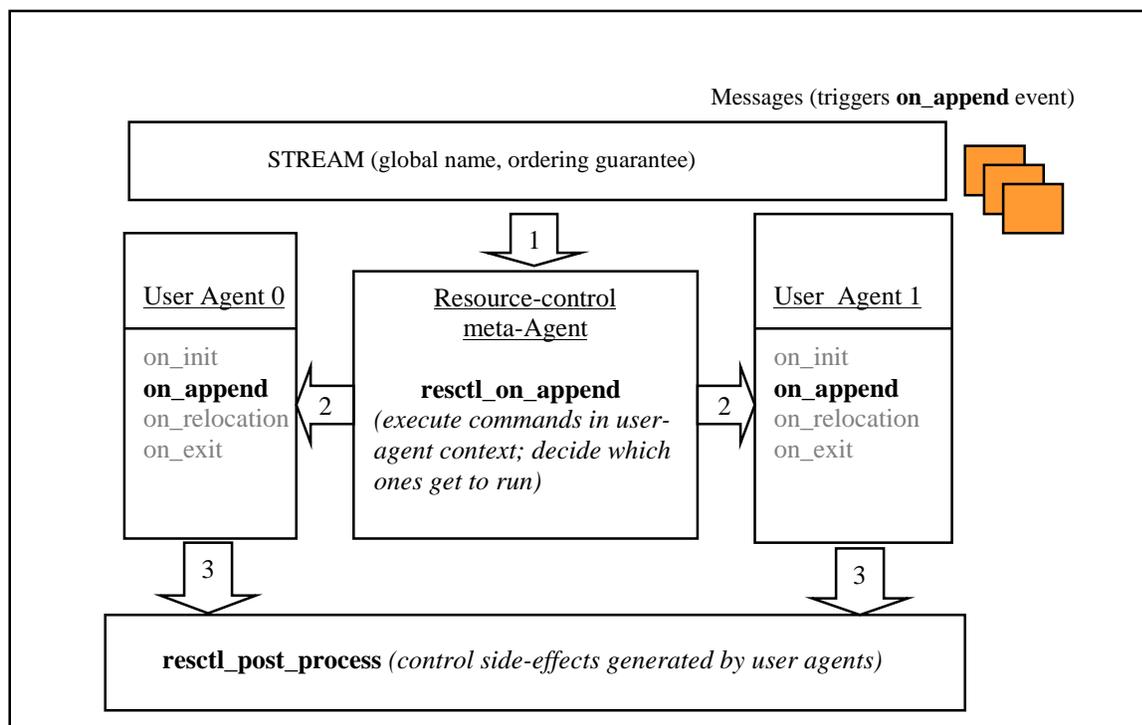


Figure 1: Stream Resource-control architecture.

The resource-control meta-agent gets control first and has the opportunity to intervene before the other agents get to execute and after agent execution has completed.

Two kinds of resource-control hooks are provided. First, each *Agent-daemon* can be started with a *Daemon Resource-control Script*. This script is specified in sections with each section corresponding to a particular

event that may result in resource-consumption for the Agent Daemon. System-level actions such as obtaining permission to add a new *Agent Daemon* or adding a *Stream* to the system are approved or denied by the *System Resource Controller* which runs at the secure, reliable location. At the level of individual *Streams*, each *Stream* may be created with its own resource controller script – which is again specified in sections with each section corresponding to a different event. The per-stream resource controller script runs as a “*meta-agent*” - it gets control before “user-level” agents get to run on each *Stream*-specific event. Using this mechanism, the system designer may place restrictions on which *Agents* get to see the message contents, the set of commands that a user-level *Agent* may use and so on. The *meta-agent* may also execute commands in the environment of the user-level *Agent* – a feature that is useful in the construction of debuggers.

After the user-agents have finished execution, a post-processing resource-control script may be registered that can restrict the side effects caused by the user-registered agents – for example by restricting movement to “dangerous locations”. We have also incorporated an I/O control and stream-flow control mechanisms for control operations and stream synchronization –details of which we omit for brevity.

Figure 1 illustrates how the resource-control mechanism works.

### 2.2 Inter Stream Communication

Although we provide RPC as an option, it is expected that the major mode of operation will be through asynchronous messages. Asynchronous messages provide the means for latency hiding which is necessary for improved performance in high-latency environments such as the Internet. The system provides a *FIFO* ordering guarantee. With this guarantee, if an *Agent* of a *Stream a* from location *x* appends a message “1” to a *Stream 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”>. This guarantee must be preserved even when both sender and receiver are moving.

### 3. An Adaptive Collaborative Toolkit

The purpose of this application is to provide a means whereby any arbitrary Tcl/Tk application may be collaboratively shared using the “What You See is What I See” (WYSIWIS) paradigm. The novel feature here is the use of dynamic re-configuration for latency reduction. The architecture of the application is as shown in Figure 2 below:

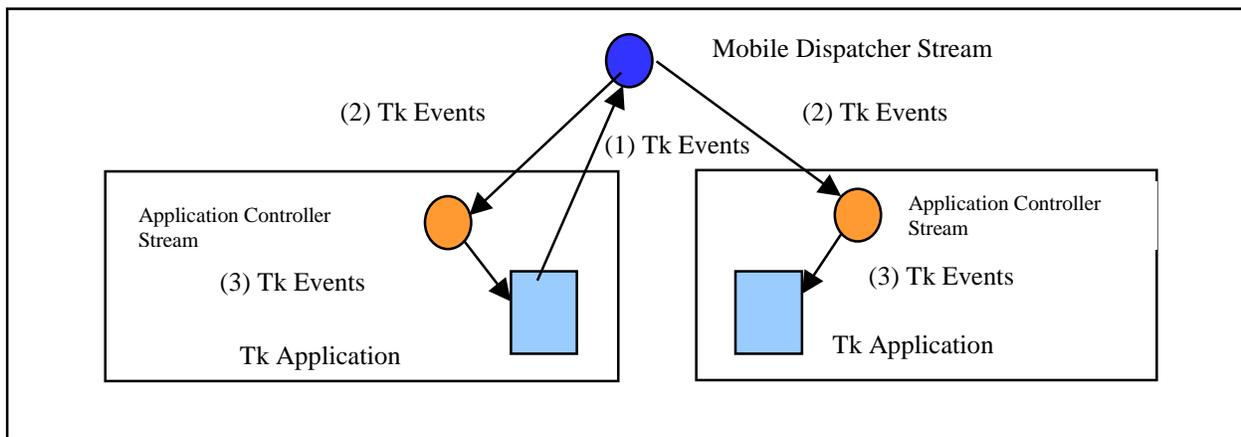


Figure 2: Tk- Collaborative Toolkit Architecture

Events are sent to the dispatcher stream first. The dispatcher stream uses mobility to reduce the expected latency for the interactive user by relocating itself to the place from where the last 50 appends originated.

Each participant runs a copy of the Tk application being shared. The application works by “stealing the callback” from each Tk widget and dispatching each Tk event generated locally to every other participant. In order to ensure WYSIWIS, each participant must receive Tk events to replay on his application in the same order. This is achieved by sending these events to an *Event Dispatcher* from where it is sent each participant as shown in the figure above. If the event dispatcher were stationary, this could become quite irritating for the interactive user as he experiences a round-trip latency for each input event. The 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.

## 5. Related Work

The design presented in this paper has been influenced by several other systems such as *TACOMA*[1], *Voyager*[2] and *Aglets*[3] as well as the first author’s experience with the *Sumatra* system[4]. Our main innovations are to provide a clear naming and inter-stream communication model, resource control mechanisms and separation of naming and location from functionality – features that we believe eases the task of building distributed systems. Our work has also been influenced by the work on *Active Networks* [5].

## 6. Future Work

There are several aspects of our system that we are currently refining. We are working on utilizing reliable multicast in our system to improve scalability. We are also working on fault-tolerance and security. We have developed an integrated Agent monitor and Streams debugger with a global break-point capability. We are working on a collaborative (peer-to-peer) cache management application involving digital microscope images and on a distributed test management framework for collaborative systems. We are also looking at inter-operability with commercial tools.

## 7. Acknowledgements

This work was supported by DARPA funding. Several aspects of our design are an evolution of ideas and experience gained from previous work with Anurag Acharya of UCSB [4]. We are indebted to Charles Crowley for making the source code of *Tk Replay* [6] freely available.

## 8. References

- [1] 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.
- [2] “Voyager White Paper”, <http://www.objectspace.com/voyager>
- [3] B. Venners, “Under the Hood: The architecture of Aglets”, *Java World*, April 1997.
- [4] M. Ranganathan A.Acharya S.D.Sharma and Joel Saltz, “Network-aware Mobile Programs”, *USENIX 97*.
- [5] D.J. Wetherall and D.L. Tennenhouse, “The ACTIVE IP option”, *Proc. ACM SIGOPS*, 1996.
- [6] C. Crowley, “Tk-Replay: Record and Replay in Tk”, *USENIX Third Annual Tcl/Tk Workshop*, 1995.