Pp. 75-82 of the *Proceedings of LISA '02: Sixteenth Systems Administration Conference* (Berkeley, CA: USENIX Association, 2002).

Stem: The System Administration Enabler

Uri Guttman – Stem Systems, Inc.

ABSTRACT

Stem is a system administration "enabler." It is not an administrative tool, but rather a general-purpose development framework that allows an administrator to craft tools to perform a wide variety of tasks in a distributed environment easily and quickly. Many common tasks can be performed with Stem scripts involving a few lines of declarations. Current example applications include log file collating and service load balancing. Using Stem, a non-programmer can craft reliable network software in a few lines of declarations that would require hundreds or thousands of lines of code in a traditional programming language such as C.

Introduction

It is a bit difficult to begin describing something that is rather unique. Stem is not an administrative tool, nor does it support a particular kind of administrative practice. It is instead a "framework" for declarative construction of networked software for a variety of purposes. Its intent is to make network software that is as easy to create as it is to describe. In many cases, Stem allows a system administrator to create such tools without learning to program by basing them upon a few templates that describe common network applications.

Stem's components include:

- A declarative configuration language by which one can define application components, known as "Cells." Stem is written in Perl, and the current configuration directives use Perl syntax.
- A runtime daemon that creates and executes components (cells) based on the defined configuration.
- A set of modules that implement useful, commonly used cells.
- Additional utility modules and command tools.

While it remains a general-purpose programming framework, Stem's primary goal is to help system and network administrators solve their problems with less work. The is the concept of "enabling" which will be a theme throughout this paper. Stem enables system administrators to easily glue existing systems together with network connections, create new networked applications with much less coding than before, and configure solutions to many common problems without any coding at all.

Stem is a general purpose system that can be used in many situations and problem spaces. By using the existing Stem modules and example configurations you can focus on your own problem space and issues and ignore many common network programming problems such as event handling and client-server and other interprocess communication. If your problem space is large, Stem enables you to cover that entire space under one architecture, thereby simplifying your design, coding and maintenance.

To Code or Not to Code

While most administrators write at least the occasional script, many (perhaps even most) do not have the time or skills to develop full-blown network applications from scratch. Stem enables both groups of administrators equally. Non-programmers can create Stem configurations or modify the supplied examples without any coding needed to solve their specific problems, simple or complex. Administrators who know Perl can create modules for functions not provided by existing Stem modules and still take advantage of the infrastructure and network services that Stem offers, allowing them to limit their programming to just their specific task. The advantages of this approach, with respect to simplifying and speeding up application creation, are obvious. A site can even split up the work, with a developer creating new Stem modules, and an administrator creating the configuration to drive and deploy them.

Stem Networking

Gluing together disparate existing applications is a common and difficult problem when attempting to automate system administration tasks. Applications can have command line interfaces (CLI), be client/server based, or use common protocols (HTTP, SMTP, etc.). Stem enables an elegant solution to this task. Stem can be used to wrap each existing application in a module along with a new, message-based interface (essentially, an API). However, there is no need to predefine message queues or compile parameters as you do in many other message-passing or RPC type systems. Once this is done, Stem declarations invoke the module under conditions you specify.

This approach allows the task of gluing together applications to be divided into two phases: first, creating a wrapper with a message-passing interface and second, using the new module within Stem, allowing the cells to communicate automatically with one another. The first task is performed only once, and the resulting module is reusable, and generic, while the second task allows the module to be quickly put to work on a specific task.

Related Work

Stem is a unique integration of several kinds of technology, but has its roots in several other tools that contain part, but not all, of its capabilities.

Message-passing Systems

First, Stem is a "message-passing system," but applying that name implicitly limits it more than is appropriate. For a programmer, the term "message-passing" generally refers to parallel programming libraries such as PVM and MPI [mpi]. Stem differs from such facilities in that it allows network applications to be created at a conceptually "higher" level and handles all of the lowest level message-passing functionality transparently to the application creator. Also Stem's messages can be sent to any cell in the current application, whether in the same hub (process), system or to a remote site. In fact, changing where a message is sent usually amounts to simply editing an address in a configuration file with no coding involved.

Another common use of the term "message-passing" is to refer to a class of commercial products commonly called Message Oriented Middleware (MOM). These products include guaranteed message delivery among their capabilities, which also typically include access to databases and transaction systems and other related services. Such products are usually targeted to the financial and business community and are rarely used by administrators and network developers. A few of the more well known MOM products include IBM's WebSphere MQ Family and Microsoft's MSMQ [mom].

Stem differs from MOM applications in several ways. First of all, MOM systems are designed for use by professional application programmers and usually require substantial programming expertise to use. In contrast, Stem is designed for use by administrators for administrative tasks while minimizing required programming. Secondly, traditional MOM systems are extremely large and entail considerable overhead, from both a computing and a staffing point of view. MOM systems typically need a database system to function, and some MOM vendors (e.g., IBM) recommend at least one full time staff person dedicated to running them. Stem is at the other extreme in that it is extremely lightweight. Finally, while Stem is Open Source, existing MOM applications are commercial products which are both expensive and proprietary.

Administrative Tools

Second, Stem is intended as an "administrative tool" but has almost nothing in common with existing administrative tools such as Cfengine [cfengine]. These tools are intended primarily to control the configuration of hosts within a network. Stem is instead intended to allow flexible interoperation between tools within a network. While Cfengine provides network communications layers so that hosts can exchange information, this information is limited to facts relative to host configuration. It cannot, for example, hand off a service request from one host to another, an operation that is trivial in Stem.

The Swatch package [swatch] overlaps to some extent with one of Stem's modules. This package monitors log file contents and searches for specified patterns set in its configuration file. The Stem::LogTail module performs a very similar function.

Monitoring Tools

Stem is also not a monitoring tool. It is better to say that Stem is a framework that makes it easier than ever to create applications which collect any desired system and network data. Thus, it can subsume many of the data collection capabilities of these tools. Stem could also be used to feed data to existing monitoring tools like RRDTool [rrdtool] or Cricket [cricket], enabling the administrator to extend their capabilities while continuing to take advantages of these tools' mature visualization capabilities.

Other Facets

Stem has something in common with many other tools and approaches. Stem's ability to function as an application wrapper has its roots in many other message-passing and "screen-scraping" systems. Its basic philosophy of creating a distributed configuration engine that responds to flexible events was first documented in Distr [distr], though this mechanism was intended solely for file distribution.

Stem Architecture

To encompass so many facets of other work, Stem has evolved a unique architecture based upon a biological metaphor of "Stem Cells." A Cell is the fundamental building block for a network application. In a running Stem application, one or more cells exist as objects in a Stem "hub"; a hub corresponds to a single daemon process. Multiple hubs can run simultaneously, and they communicate with one another via constructs called "portals" that use TCP/IP sockets. Hubs can communicate with other hubs running on the same system or any network-accessible host, with Stem handling all of the interprocess communication.

Stem is a fully event driven system. Events can be any of the common network operations such as socket connections, I/O on character devices (terminals, sockets, pipes, etc.), and timers. Stem uses a consistent technique for the delivery of messages based on all kinds of events.

Cells

Stem cells are addressable objects which can send and receive messages. Cells are registered at creation time by name. There are three kinds of Stem Cells:

- **Class Cells** correspond to a Stem hub-wide (process) resource. These Cells are usually self-registering and generally use the class name as their identifier but more intuitive aliases may also be defined.
- **Object Cells** are application global objects. Most often, they are created and registered by the configuration which the Stem system is running, but they can also be loaded or created at runtime. They are generally long lived and last as long as the Stem hub is running.
- **Cloned Cells** are additional instances copied from an existing parent object cell. They share the parent's Cell name but are additionally given a unique target name which assigns them a unique address. Cloned cells are similar to what other systems would call sessions. They are dynamically created upon request and last only as long as needed.

The heart of Stem is the messaging subsystem and the heart of that is the registry. This is where all knowledge of how to address cells is located. Each cell is registered by its name and if it is a cloned cell, also by its target name, and messages are directed to it via these names.

Messages

Stem Messages are how Cells communicate with each other. Messages are simple data structures with two major sections, the address and the content. The address contains the Cell name that the message is directed to and which Cell sent it. The content has the message type, command and data.

Message addresses are name triplets of Hub/Cell/Target. The Cell name is required and the Hub and Target are optional. These triplets form globally unique addresses with the overall Stem system.

The Message address section has multiple address fields. The two primary fields correspond to the common email headers and are called 'to' and 'from'. The 'to' address designates which Cell will get this message, and the 'from' address says which Cell sent this message.

The Message content has information about this message and any data being sent to the destination Cell. The primary attribute is 'type' which can be set to any string, but common types are data, cmd (command), response and status. Stem modules and Cells can create any Message types they want. The other major attribute of the content is the data, which holds a reference to the message data.

Modules

The various Cells classes are implemented as Perl modules, and many useful Cell types are included with the Stem package. These are among the most important:

- Stem::SockMsg: Cells that connect to/accepts connections from sockets. These cells function as a socket-to-Stem message gateway.
- Stem::Switch: Multiplexes Stem messages to multiple destinations according to maps which can be dynamically modified.
- Stem::Portal: Manages connections between Stem hubs, facilitating message transmission across the network (including authentication and security functions).
- Stem::TtyMsg: Provides a TTY interface to a Stem hub.
- Stem::Proc: Creates Cells that fork external processes and manages them.
- Stem::Log: Writes and manages Stem logs (which may be associated with external files).
- Stem::Log::Tail: Monitors active external files (typically log files), sending newly acquired data into the Stem logging subsystems on either a periodic basis or on demand.
- Stem::Cron: Creates and manages scheduled message submissions. Such messages can be sent anywhere in a Stem network and can trigger any Stem operation. If the message is addressed to a Stem::Switch Cell, it can then be sent to multiple destinations and trigger events across the network from a centralized schedule.

In addition, Stem includes other utility modules which provide services to active Cells. These services include asynchronous I/O, cloning of cells, flow control or local and remote method calls and logical pipes.

Configurations

Stem differs from all other networking toolkits by being architected around configuration rather than software. A configuration file instructs the Stem engine which Stem cells to construct and register. When you invoke Stem, it interprets this configuration and creates cells as needed without any need to compile networking code.

Configuration files are structured using Perl objects. Each desired cell is specified as a Perl object with a list of attributes. Several examples are discussed in the next section.

Example Application: An inetd-like Server

We will consider a few versions of a small Stem application, chosen for its use of typical Stem components as well as in response to the space limitations imposed on this paper. None of them require any programming on the part of the system administrator.

A Simple First Version

This version of the application serves as a good starting point for understanding Stem. It uses only existing Stem modules to create an application which can execute a process on a local or remote system. As such, the only task required to create the application is to set up a file containing the Stem configuration. The simplest version is given in Listing 1.

This configuration uses three cells:

- A Stem:TtyMsg Cell. This Cell is used to enable commands for the hub to be typed in at the keyboard. In this configuration the default attributes are used.
- A Stem::Proc Cell named "mon". This section of the configuration file will create a Cell that starts a process on demand. Here we specify a list of arguments for the Cell: the path to the command to be run, and some generic Cell attributes. The latter specify that the cell is to be cloned and that it will only send the data from its process when it exits.

```
# uptime.stem
[
   class => 'Stem::TtyMsg',
   args  => [],
],
[
   class => 'Stem::Proc',
   name  => 'mon',
   args  => [
      path      => '/usr/bin/uptime',
      cell_attr => [
         'data_addr' => 'A',
         'send_data_on_close' => 1,
      ],
   ],
],
[
   class =>  'Stem::SockMsg',
   name  =>  'A',
   args  =>  [
      port      => 6666,
      server    => 1,
      cell_attr => [
         'data_addr' => 'A',
      ],
   ],
],
```

Listing 1: Simplest configuration file.

- A Stem::SockMsg Cell named "A". This Cell will listen on a socket (the port address is specified by the port attribute). When a connection request is accepted, it will create a logical pipe to the 'mon' cell as specified in the 'pipe_addr' attribute.

In this example when a socket connection is made, the logical pipe to 'mon' is created, which causes the 'mon' cell to clone and fork the uptime program. Its output is collected and then sent back to the 'A' send and then on to the socket. Stem will take care of creating and sending all of those messages automatically.

Running this configuration requires the following commands:

```
$ xterm -T Stem -n Stem \
           -e run_stem uptime
$ xterm -T Monitor -n Monitor \
      -e telnet localhost 6666
```

We use two xterm commands to make Stem's operations visible. The first command starts the Stem hub daemon process and attaches the Stem::TtyMsg Cell to it so commands can be entered. The second command attaches a second window to port 6666, the Stem::SockMsg Cell, using a telnet command. Figure 1 illustrates the resulting windows.

In the "Stem" window, we send the cell_trigger command to the mon Cell. This causes the Stem::Proc Cell to execute its command. Note that the output appears in the "Monitor" window attached to port 6666 each time the cell is triggered.

A Piped Version

The problem with the example above is that you must enter a 'cell_trigger' command to make the process execute and only one telnet session can be used. This new version (see Listing 2) will make the process execute when the telnet connects to the socket. Note it uses the 'pipe_addr' attribute which will create a logical pipe between the cell 'A' and the 'mon' cell. Actually the 'mon' cell will be cloned when a pipe to it is created and that cloned cell will run the process.

To run this example, name the configuration file up3.stem and run this command:

```
$ xterm -T Stem -n Stem \
           -e run_stem uptime2
```

Then, in another window run the telnet command:

```
$ telnet localhost 6666
```

Each time you run telnet you will invoke the uptime command and see its output from telnet which will then exit.

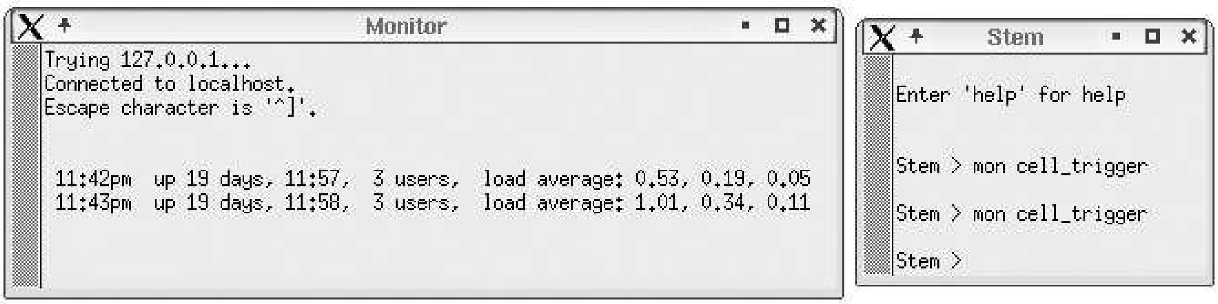

Figure 1: A simple Stem application.

A Multi-Hub Version

The previous two Stem applications were extremely simple, but their potential functionality is very powerful. They can be extended in several ways:

- The process can be executed on a different host than the triggering host.
- The process can be executed on multiple remote systems.
- The output can be sent to more than one destination: multiple sockets on different systems, a log file, another Stem cell for further processing, and so on.
- A different command or program can be run. The uptime command merely serves as a proof-of-concept here. Any command that is needed could be executed.

```
#uptime2.stem
[
   class => 'Stem::TtyMsg',
   args  => [],
],
[
   class => 'Stem::Proc',
   name  => 'mon',
   args  => [
      path      => '/usr/bin/uptime',
      cell_attr => [
         'cloneable' => 1,
         'send_data_on_close' => 1,
      ],
   ],
],
[
   class =>  'Stem::SockMsg',
   name  =>  'A',
   args   =>  [
      port      => 6666,
      server    => 1,
      cell_attr => [
         pipe_addr => 'mon',
      ],
   ],
],
```

Listing 2: Improved with concurrent sockets.

- More than one command can be supported by defining multiple Stem::Proc cells. In this way, the application can function in a similar way to inetd in that it can start any one of a number of preconfigured servers upon demand.
- The process can be triggered other ways than by manually entering a command or by connecting to a socket: by a different Stem cell, according to a schedule (using Stem::Cron), etc. The triggering can come from the server hub, the client hub, or elsewhere in the Stem system.

Listings 3 and 4 illustrate a multi-hub version of this application. Notice how easy it is to split the simple version into an implementation which can be run across the network.

To run this application, start processes like these (as before):

```
$ xterm -T Server -n Server \
        -e run_stem uptime_server
$ xterm -T Client -n Client \
        -e run_stem uptime_client
```

Then, in another window run this command:

```
$ telnet localhost 6666
```

This will behave the same as the uptime2 example but it is split over two hubs. Notice that other than adding the Hub and Portal cells, the only change to the configuration was adding a hub name to the 'pipe_addr' attribute in the uptime_client configuration. This illustrates how easy it is to distribute applications written in Stem across a network. Sending messages to local or remote cells is done the same way – typically only the address will need to be changed.

This example application and its variations provide some insight into Stem's flexibility and capabilities. The Stem distribution comes with several other example applications and a cookbook that shows you how to create your own cells.

```
#uptime_server.stem
# Name the hub so the client can
# refer to it.
[
   class => 'Stem::Hub',
   name  => 'uptime_server',
   args  => [],
],
[
   class => 'Stem::TtyMsg',
   args  => [],
],
# Set the portal to be a server:
# listen for portal connections from
# any host the port defaults to 10000
# but can be set here
[
   class => 'Stem::Portal',
   name  => 'listener',
   args  => [
      'server'  => 1,
      'host'    => '',
   ],
],
[
   class => 'Stem::Proc',
   name  => 'mon',
   args  => [
      path      => '/usr/bin/uptime',
      cell_attr => [
         'cloneable'          => 1,
         'send_data_on_close' => 1,
      ],
   ],
],
```

Listing 3: Multi-hub server.

Critique and Analysis

The major infrastructure work of creating Stem has been completed, and the package is working well where it has been deployed. The design has proven to be as flexible as was intended, and Stem applications have been created by administrators unfamiliar with the package after just a few hours.

Stem's planned extensibility has also been verified in that administrators have successfully written additional Stem modules in Perl and integrated them with those that the package provides.

Stem's highly modular design has been proven to be instrumental in extending it. New modules can easily be created and integrated. Internal services have been developed and quickly used by other modules and cells. Its simple message-passing API allows almost any external application, service or protocol to interact with any other.

```
#uptime_client.stem
# Name the hub so server can refer to it.
[
   class => 'Stem::Hub',
   name  => 'uptime_client',
   args  => [],
],
[
   class  => 'Stem::TtyMsg',
   args   => [],
],
# Create a client portal.
# this will connect to the portal
# in the 'monitoring' hub the default
# host is 'localhost' but it can be
# set here the port defaults to
# 10000 but can be set here
[
   class => 'Stem::Portal',
   name  => 'server',
   args  => [],
],
[
   class  =>  'Stem::SockMsg',
   name   =>  'A',
   args   =>  [
      port      => 6666,
      server    => 1,
      cell_attr => [
         cloneable => 1,
         pipe_addr => 'uptime_server:mon',
      ],
   ],
],
```

Listing 4: Multi-hub client.

Stem's configuration file format currently takes the form of Perl data structures. This has performance implications and security issues as well as presenting a somewhat eccentric interface to non-Perl literate system administrators. In the future, we plan to support multiple configurations formats including XML.

Similarly the format of Stem's message (when serialized over a pipe) also needs to support other formats. But as with the configuration formats, it is just a matter of having modules that can convert the internal message structure to/from an external format. This will allow other systems to be more easily integrate as they can then send/receive Stem messages.

Stem's security support currently is weak. We have demonstrated to ourselves that we can use ssh for message-passing between Stem processes but the design was not good enough for production. We have plans to redesign it to be integrated with Stem's socket module so that any IPC (not just message-passing) can use it. Also the design would allow a choice of secure transport (ssh, SSL etc.) by using the same modular plug-in design as mentioned above.

Future Work

Stem is extremely modular in design and can be extended easily in many directions. Here is a short list of some items that are in our development queue now:

- State Machine: A text based state machine that will take input from multiple sources: socket, processes and messages. It will have many features including state callbacks, input and output buffers, regular expression matching, and the like.
- Flow Control: A module that will allow a Stem Cell to control the logic flow of method calls, regardless of whether they are local or remote. It manages a combination of synchronous (local) and asynchronous (remote via messages) object method calls in a simple mini-language that will have the common flow control operations such as IF/ELSE, WHILE, etc. This greatly simplifies the task of coordinating distributed operations upon an object (a Stem Cell) such as accessing a database or using subprocesses and remote protocols.
- Network Protocols: When the State Machine is finished, it will be used in various protocol modules which will enable Stem to communicate with programs which use the popular protocols such as HTTP, FTP, SMTP, etc.
- GUI-based Configuration Tool: A longer term goal is to integrate Stem with a GUI toolkit such as Tk or Qt in order to develop a tool for creating and visualizing Stem configurations. Doing so will also make it possible to create a wide range of GUI front ends for Stem based applications.

Availability

Stem is Open Source software, it is licensed under the GPL and is available without charge from Stem Systems. Our website is http://www.stemsystems.com .

Acknowledgments

I wish to thank Aeleen Frisch, Alva Couch, and Will Partain for their help with this paper.

Author biography

Uri Guttman graduated from MIT in 1983 with a B.S. CSE. He has been developing software for 25 years. Stem is the result of his extensive experience in systems architecture, networking, communications, API design and Perl. He is currently president of Stem Systems and can be reached at uri@stemsystems.com .

References

[cfengine] Burgess, Mark, "Cfengine: A Site Configuration Engine," *USENIX Computing Systems*, USENIX, 1995.

[cricket] Allen, Jeff R., "Driving by the Rear-View Mirror: Managing a Network with Cricket," *Proceedings First Conference on Network Administration*, USENIX 1999.

[distr] Couch, Alva L., "Chaos Out of Order: A File Distribution Facility For 'Intentionally Heterogeneous' Networks," *Proceedings LISA 1997*, USENIX, 1997.

[mom] http://www-3.ibm.com/software/ts/mqseries/ and http://www.microsoft.com/msmq/default.htm .

[mpi] The Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard" and "MPI-2: Extensions to the Message-Passing Interface," http://www.mpi-forum.org .

[rrdtool] Oetiker, Tobias, "RRDTool," http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/manual/index.html .

[swatch] Hansen, Stephen E., and E. Todd Atkins, "Centralized System Monitoring With Swatch," *Proceedings LISA 1993*, USENIX, 1993.