SystemStarter and the Mac OS X Startup Process

Wilfredo Sánchez
wsanchez@mit.edu

Kevin Van Vechten
kevinvv@uclink4.berkeley.edu

Abstract

This paper documents a program in Mac OS X called SystemStarter. SystemStarter brings the system from its initial state up to a state where basic services are running and a user may log in. It replaces the previous `/etc/rc` startup sequence employed by Mac OS X's predecessors in order to address some additional goals set forth by the Mac OS X project.

SystemStarter is part of the BSD subsystem in Mac OS X, now known as Darwin, though it's creation predates Darwin as an open source project; it was therefore created by a single author, though it now enjoys several contributors.

Background

Mac OS X is the latest version of Apple's operating system for its Macintosh computer line. One of the unique attributes of Mac OS X as compared to previous versions of Mac OS is the use of BSD as its base system implementation. During the Mac OS X operating system bringup, a great deal of effort was spent on integrating the many facilities of Mac OS on this new BSD foundation [1]. For the most part, BSD facilities did not have to change significantly in order to accomplish this. One of the few that did is the system startup sequence.

Mac OS X started largely form the code base for a prior OS product called Mac OS X Server, once better known by its code name "Rhapsody." Rhapsody uses a startup sequence inherited from OpenStep. The kernel launches init, which runs a script `/etc/rc` as per BSD convention. `/etc/rc` would then run scripts in the directory `/etc/startup` in lexical order based on filenames. The script `0100_VirtualMemory` would run before the script `1600_Sendmail`, for example. When these scripts finish, `/etc/rc` would exit and init would then bring up the multiuser login prompt as configured in `/etc/ttys`.

The Problem

This mechanism was simple, and by allowing separate scripts for each service to be run at startup, it allowed users to insert services into the startup sequence in a straightforward manner. However, there were several drawbacks:

• The lexigraphic ordering is fragile. If we were to change the order in which scripts run, or insert or delete a new standard service into the sequence, we may have to renumber several of the files. This means that a user (or a vendor package installer) would have to place additional scripts into different locations in the order depending on the system version, and it would be impossible to know that the future system releases would not again break the ordering.

• The startup sequence is inherently limited to serialization by this design. It may make a lot of sense to run a disk-intensive task such as cleaning up `/tmp` while simultaneously running a network-blocked operation such as requesting a DHCP lease. In the `/etc/startup` scheme, these were always run in sequence, failing to take advantage of the systems ability to manage multiple resources simultaneously.

• It was not easy to know which scripts were installed by the user, and which are part fo the system software, as they are co-mingled.

• /etc is hidden by the Finder. A user should be able to copy and manage startup scripts at a well-known location, but /etc, /var, /usr, and other Unix directories are by default hidden from view.

• Graphical startup was difficult. The window system must start very early in order to display startup progress, and the startup scripts would need some way to update the display. Rhapsody used a command-line tool which could draw directly to the display's frame buffer before the window system was running, but this was difficult to maintain and highly limiting. You could not, for example, localize text, as font support wasn't available, etc.

Alternatives

Other BSD systems were using schemes that we similar, usually involving a couple of scripts in addition to /etc/rc which were specific to networking, loading kernel modules, etc.

System V systems (such as Solaris, and in it's own quirky way Linux) had a fairly different mechanism that was worth studying.

System V uses separate scripts similar to our /etc/startup scripts, but each script has the ability to stop as well as start a service, which provides for the possibility of a shutdown sequence and the ability to start and stop services after the system is running as the user needs them [2]. Extending /etc/startup to allow this would be fairly easy, and the value of doing so was clear, so it quickly made it to the list of features to include in a new scheme.

Another feature in System V startup is the concept of runlevels, which represent different system states [3]. Single user mode is one runlevel, multiuser mode is another. There are other runlevels for intermediate states, as well as pseudo-runlevels for shutdown and restart. This concept didn't last long on the feature list. The value of having runlevels is questionable at best, their semantic meaning is vague (which is not helped by their being labeled with numbers instead of names), they add a significant degree of complexity in managing startup scripts (Which runlevels provide which services? How do you handle transitions?), and explaining all of this to a non-technical user (in fact, even a technical user) is not a simple exercise.

It should be noted that NetBSD began work on a replacement for its /etc/rc [4] shortly after work on SystemStarter had begun. Wilfredo Sánchez, who had started the work on SystemStarter, passed along his current design ideas, but though the goals overlapped, the projects had already gone down different design paths and not much collaboration came of it.

Solutions

The initial design was to create a new program, called SystemStarter, which will be responsible for managing the startup process. Services will be described by "startup items," which know how to start a service, the service's dependencies, and other information such as user-visible strings. SystemStarter will compute an order for starting items based on the dependencies and see that the items are run accordingly.

SystemStarter is implemented in C using Mac OS's CoreFoundation framework, which provides a number of useful data types (strings with encoding support, arrays, hash tables) and functionality (XML parsing, interprocess communication, run loop, object reference counting) in as object-oriented a fashion as allowed by the C language.

Startup Items

Startup items are comprised of several parts, and implemented as a directory containing several well-defined files. Each file addresses a specific part of the item's functionality. In Mac OS parlance, this is termed a "bundle" directory. In the application toolkits, bundles are often treated as single objects, similar to files.

As with other startup mechanisms, a program, usually a shell script, describes the activity required in order to start the service(s) provided by the item. The script is given a "start" argument during system startup. In the future, the "stop" and "restart" arguments will be sent to the script at other times. (See "Ongoing Work" below). /etc/rc.common provides some useful functions (for example, the GetPID function used here gets a process ID registered in /var/run, if available) as well as a standard mechanism for handling arguments passed in by SystemStarter. For example, a script for starting cron might look like:

```
#!/bin/sh

. /etc/rc.common

case $1 in
 start)
   ConsoleMessage "Starting cron"
   cron
   ;;

 stop)
   if pid=$(GetPID cron); then
       ConsoleMessage "Stopping cron"
       kill -TERM "${pid}"
   else
       echo "cron is not running."
   fi
   ;;

 restart)
   ConsoleMessage "Restarting cron"
   if pid=$(GetPID cron); then
       kill -HUP "${pid}"
   else
       cron
   fi
   ;;

 *)
   echo "$0: unknown argument: $1"
   ;;
esac
```

In addition to a procedure for starting its service(s), a startup item needs to provide enough information so that it can calculate an order for all of the startup items. This information is described in a property list file in the item. The format of the file is either a NeXT property list, or XML. (XML is the preferred format for preference files in Mac OS X, and can be edited with a tool like PropertyListEditor, but we will use the NeXT format here because it is significantly more compact and readable, and friendly to manipulation as raw text.) And example for `cron`:

```
{
 Description     = "timed execution services";
 Provides        = ("Cron");
 Requires        = ();
 Uses            = ("Cleanup", "Network Time");
 OrderPreference = "Late";
 Messages =
 {
   start = "Starting timed execution services";
   stop  = "Stopping timed execution services";
 };
}
```

A human-readable description of the item is presently used in debugging only, but may also be used by a startup manager application in the future. Strings are also provided for display to the user when the item is starting and stopping. In the future, these strings will be passed by the script via IPC to SystemStarter, rather than hard-coded in the property list, which will allow for more accurate progress indicators.

A list of services provided by the item is noted, for use by dependent items. For modularity, it is generally best that separate services each reside in separate items, but there may be cases where a single server is used to provide more than one service, therefore a list is allowed. Similarly, a list of services on which the item is dependent is given. These fall into two classes. Some services may be required, in which case, the SystemStarter will not attempt to start the item unless the requirements are started first (which may mean the item never starts). Other services are desired, but optional. For example, cron is a time-based service, which makes it desirable that the computer's clock be synchronized with the network before starting cron, but failure to do so should not prevent cron from starting at all. In this example, cron also wishes to wait on system cleanup, and has no hard requirements.

CoreFoundation's property list parser is used to read the property list and generates an object graph using its CFDictionary, CFArray, and CFString object types. The items are represented in memory as CFDictionary objects. The original plan was to generate a new object graph directly representing the dependency tree for fast searching of dependents. However, in the current implementation items are simply stored in a CFArray and searched linearly. Whenever an item's script exits, all remaining items are checked to see if their dependencies are now met in light of the newly available service(s). The code to manage an array is much simpler, and given that the number of startup items on a system is not expected to be large, the anticipated performance benefit of generating a dependency tree in memory is negligible.

Additional files in the startup item contain localized versions of the strings provided for user-visible display.

Filesystem Layout

Traditionally System V [5], Linux [6], and BSD systems search for startup scripts in a subdirectory of `/etc`, such as `/etc/init.d`. This is undesirable for SystemStarter because the /etc directory is hidden from view in Mac OS X, and because it does not allow third-party startup items to be easily distinguished from the standard items provided by the system distribution. In order to address both of these issues, SystemStarter searches for startup items in one of several directories. Startup items provided with the system distribution are stored in the `/System/Library/StartupItems` directory; these are said to be in the "system domain." Users and third-parties are encouraged to place items in the `/Library/StartupItems` directory, known as the "local domain." Items in the local domain are not deleted or replaced when the system is upgraded. Furthermore, if an item in the local domain provides the same service as an item in the system domain, the item in the local domain takes precedence. In this way, a user may supersede a standard item without worrying about the behavior reverting after a system upgrade.

Work is in the design stage to provide support for a "network domain" which could be used to ease remote administration of many machines. After mounting a site-local NFS filesystem, SystemStarter could be prompted to search for startup items in `/Network/Library/StartupItems`. The network domain would take precedence over the system domain, but not the local domain.

Graphical Startup

The majority of Unix-variant systems boot in a text console and print out significant debugging information as the system starts up. While this is useful at times, it can be rather baffling to most consumer users. In Rhapsody, we had a program called fbshow, which could draw to the display's frame buffer directly during startup, after which we would start the (PostScript) window server. It would draw a progress panel on screen and could print status text in one font. This was inflexible in that the graphics it used were compiled into the binary, and text was not internationalizable (eg. no Japanese fonts). In Mac OS X, the window server was far more lightweight, and could be started very early in the startup process. This gave full font support, plus all of the display features of CoreGraphics (a.k.a. Quartz), such as PDF rendering and compositing.

It should be noted that because SystemStarter boots the system, its failure due to a crash can be catastrophic to the user. The more API the program draws on, the more libraries need to be loaded, and the greater likelihood of failure due to something like a corrupt file on disk. This is particularly relevant to SystemStarter, because *it* will be responsible for running fsck, which checks for corrupt files and repairs them. For this

reason, rather than making the SystemStarter dependent on CoreGraphics framework (and whatever CoreGraphics depends on), the built-in display functionality in SystemStarter is text based and the Quartz code is placed in a loadable module. If the module fails to load, SystemStarter falls back to text-mode. This also enables additional modules to be written, which proved useful in Darwin, where an X11 module can provide graphical boot using XFree86.

Ongoing Work

Written by Wilfredo Sánchez as an employee of Apple, SystemStarter was first publicly released in Darwin 1.0 and remained mostly unchanged through the release of Mac OS X 10.0 (which corresponded to the Darwin 1.3 release). It successfully provided a user-extensible system startup mechanism, and a graphical startup consistent with the Mac OS experience. After the release of Mac OS X 10.0, Darwin Committers Wilfredo Sánchez and Kevin Van Vechten continued work on SystemStarter to provide a more complete feature set.

Starting and Stopping Services

System V and other BSD systems provide a mechanism to start and stop services after the boot sequence. This feature is valuable, and useful to the system control panels, which allow users to enable and disable services as they see fit. However, care should be taken to ensure that when services are started or stopped, dependencies are accounted for. Work is being done on SystemStarter so that an item's dependencies are accounted for automatically using the dependency graph of each item.

For example, if the user asks to enable NFS, SystemStarter should ensure that portmap is running. Similarly, if portmap is terminated, SystemStarter should ensure that NFS is shut down as well. This is a difficult exercise, as the interaction between the control panels, the service manager, and the user can become rather complicated. For example, the user should be informed that shutting down one service will result in others shutting down as well. This information is tracked by SystemStarter, but would somehow have to get to the user via the control panel. Additional work is still in the design stage which may add a way for applications like the control panels to pass useful information like this from SystemStarter to the user.

Parallel Startup

One of the goals of SystemStarter going forward is to transition from a serialized startup sequence to a more flexible parallel startup sequence. A side effect of having a serialized startup sequence is that only one service can be brought up at a time. This limitation prevents the startup sequence from taking any advantages of the operating systems ability to schedule multiple tasks simultaneously such that system resources are maximally used. For example, if the system was booted after a power failure or crash, the filesystems on disk will be dirty and must be verified before they are mounted. This process is very disk intensive and can take some time. Similarly, many systems are configured to acquire their network parameters via a service like DHCP or NetInfo. When the network is busy, this can also take some time while waiting on a response from the network service. If startup is serialized, the system must wait on the disks to be checked and then wait on the network configuration. However, there is no reason why both of these cannot happen at in parallel, so that the actual time spent during startup is only that of the longer of the two services, rather than the combined total of both.

Because SystemStarter uses a dependency graph for startup items rather than an ordered list, it is possible to implement a sequence where items run in parallel. As new services become available, each pending service can be started if its dependencies are then satisfied.

Partial Startup and Shutdown

There are times when administrators wish to boot into single user mode to debug or recover parts of the system. In those cases, it is often desirable to bring the system up to an arbitrary point in the startup sequence. While the System V approach offers runlevels to allow the system to be in one of several predefined states, the runlevels are discrete and offer no assistance when a state other than one defined by one of the runlevels is desired.

Work is underway to enable SystemStarter to allow the system administrator bring up enough of the system to provide a specified service. For example, requesting that the "`Network`" service be brought up will start up all services required to use the network, but will not start any additional services. SystemStarter uses its dependency graph to determine what the specified service's prerequisites are. SystemStarter then selects only the items which are required, and performs the standard startup sequence using this subset of startup

items.

It is also possible to take advantage of the dependency graph to providing a logical system shutdown sequence. Traversing the dependency graph in the opposite direction from startup, SystemStarter runs each script with a "`stop`" argument such that each script's dependents are stopped before the antecedent is stopped. For example, a request to stop the "`Network`" service will stop all of the services that require the network, then bring down the system's network interfaces.

Interprocess Communication

During a traditional startup sequence, startup scripts print messages to the console and system log to report their progress for logging and debugging purposes. However, during graphical boot, it is desirable to print messages which indicate to the user (at a simpler level) what is going on. An inter-process communication mechanism in being implemented so startup scripts can send messages to SystemStarter which in turn displays them during graphical boot.

During the startup sequence, SystemStarter listens for messages on a named Mach port. Messages are composed of an XML property list which specifies the type of message and its arguments. Startup scripts use a provided tool to send a message to SystemStarter which will be displayed on-screen. Each message contains a token identifying the startup item which originated it. SystemStarter then attempts to localize the message based on the localization dictionaries provided with each startup item. Finally the localized message (or the original message if no localization could be provided) is displayed.

The XML property list format for IPC messages was chosen to ensure maximum forward and backward compatibility between future versions of SystemStarter and tools that communicate with it. Mach ports were chosen for the IPC mechanism because Darwin's CoreFoundation library, which SystemStarter already used extensively to manage its internal data structures, provides a simple API for sending messages between processes using Mach ports. Additionally, Mach port invalidation callbacks are used to monitor startup item termination, allowing both process termination events and IPC events to be handled from a single event loop.

More IPC message types are planned. Specifically, messages allowing startup scripts to report the success or failure of a particular task would be useful, as items may provide multiple services and accounting for which specific services are running would improve SystemStarter's dependency tracking. Ultimately startup scripts may be required to respond to a "`status`" command, reporting to SystemStarter the status of each service the item provides. This information will be useful in determining which services the system can provide at a given time, as well as what state graphical controls should be set to when displaying system control panels.

Conclusion

In its initial release, SystemStarter succeeded in removing the fragile lexigraphic ordering of startup items, providing a graphical boot sequence, and separating third-party scripts from those of the system distribution. Although SystemStarter differs significantly from other system startup mechanisms, the startup item scripts are fairly similar to those found on other systems. Porting startup items to Darwin involves slight modifications to a startup script and the creation of a property list file describing the item. Thus, SystemStarter effectively accomplished its goals with low overhead and a minimal loss of compatibility.

Availability

SystemStarter is available in Mac OS X and Darwin systems. The source code of SystemStarter is available as part of the Darwin project at:
http://www.opensource.apple.com/

About the Presenters

Wilfredo Sánchez is a 1995 graduate of the Massachusetts Institute of Technology, after which he co-founded an Internet publishing company, Agora Technology Group, in Cambridge, Massachusetts; he then worked on enabling electronic commerce and dynamic applications via the world wide web at Disney Online in North Hollywood, California. Fred later worked as a senior software engineer at Apple Computer in Cupertino, California, primarily on Darwin, the BSD subsystem in Mac OS X, as a member of the Core Operating System group, and as engineering lead for Apple's open source projects. He continues to work on Darwin as a volunteer developer. Fred is also a member of the Apache Software Foundation, and a contributor to various other projects, including NetBSD and FreeBSD. He now works at

KnowNow, Inc. as developer community manager.

Kevin Van Vechten is an undergraduate at the University of California, Berkeley, majoring in Electrical Engineering and Computer Science. He works as a consultant specializing in custom database and networking solutions. Kevin also contributes to Darwin and other various projects.

References

[1] Wilfredo Sánchez; *The Challenges of Integrating the Unix and Mac OS Environments*; USENIX 2000 Technical Conference; San Diego, California; 2000

[2] Sun Microsystems, Inc.; *How to Use a Run Control Script to Stop or Start a Service*; System Administration Guide. Volume 1. Solaris 8 System Administrator Collection. Fatbrain, February 2000. 116.

[3] Sun Microsystems, Inc.; *Run Levels*; System Administration Guide. Volume 1. Solaris 8 System Administrator Collection. Fatbrain, February 2000. 109-110.

[4] Luke Mewburn; *The Design and Implementation of the NetBSD rc.d System*; USENIX 2001 Technical Conference; Boston, Massachusetts; 2001

[5] Sun Microsystems, Inc.; *Adding a Run Control Script*; System Administration Guide. Volume 1. Solaris 8 System Administrator Collection. Fatbrain, February 2000. 117.

[6] Red Hat, Inc.; *Behind the Scenes of the Boot Process*; The Official Red Hat Linux Reference Guide. Red Hat Linux 7.2. Red Hat, Inc., 2001.