Controlled Flexibility in System Design

A. Brady Montz and Larry Peterson
University of Arizona

1 Challenge

Historically, applications have been distributed in order to access more resources than a lone machine could supply, such as CPU power, disks, or the computers themselves (for fault tolerance). We are increasingly seeing applications which are distributed because they are meant to be used by multiple people, and multiple people generally have multiple machines. These applications are distributed by definition. A good example of this is a distributed multimedia application such as multicast video, email, or the World Wide Web. It is well understood how the interaction of software with people makes the software more complicated; we are entering the realm of software which exists purely to glue people together.

The popular focus on multimedia is on video and audio applications. While these certainly challenge current computers and profit companies, they are merely a shadow of the possibilities. This potential is not the ability of a computer to display one medium, but the ability to gracefully and efficiently handle a wide variety of data flows with differing requirements, many of which did not exist when that computer or its operating system was designed. The variation of types and topology of such flows is limited only by the creativity and whims of the human users. What sort of information flows will computers serve to us? What kinds of hardware and software will these machines need and have? How will these systems interact? We do not know. Nobody does. And we have to design our software to face this environment. This is the challenge of multimedia, and it parallels the general challenges facing these diffusely distributed applications in the near future.

The important issue is not that software needs to be flexible and easily reconfigurable. We need to be able to control, predict, and modify the behavior of our software, and this requires that we have a firm understanding of and control over how software is configurable. We should not strive to make endlessly extendable systems – rather we should strive to make systems which are exactly as extendable as required. It is in this context that we are designing the Scout[6] system.

2 Demands of Configuration

Software designers have always been challenged to keep pace with the increasing demands placed on computers. Software, once written, is released into a dynamic world. Therefore, a fundamental property of successful software is that it can be changed. Most of the attention to this has been placed on *static* changes to software, that is, the ability to modify the code base of a program. While research is being done on *dynamic*, or runtime, software changes, this area is less mature.

Since most of our knowledge about modifying software relates to static changes made by programmers, it is not surprising that this has been leveraged to affect a sort of dynamic change. This has resulted in systems like the World Wide Web, which is essentially a huge distributed application written in an interpreted language, which is modified by reprogramming. While fabulously successful, it has labored to extend past the uses foreseen by the original designers largely because doing so requires coordinated effort by most of its users.

In these sorts of huge, social, distributed applications, flexibility is vital, since these applications serve no purpose other than to satisfy people. Recent research into areas such as active networks[9][12] underscores the desire to make networks fittable to our needs, rather than those which dictate their form to us. To achieve this, we must understand the flexibility demands placed on these applications, and we must understand the nature in which software can be changed, both before and after it is run.

Here is a partial list of the demands placed on distributed applications such as the WWW, email, DNS, file systems, active networks, etc.:

- They must run on a variety of hardware and operating system environments.
- The various pieces running on different machines must cooperate with each other, even when the administrators of those machines may not.
- They often perform vaguely defined tasks which change over time.
- Their future behavior is practically impossible to predict, since it depends on large scale human activity.
- While they may not be fault tolerant or secure, their failings in those areas should be understood and predictable.

This list is daunting, especially since some of them may well be mutually exclusive. One shared trait is that the software comprising these applications should be able to react and change according to its environment in a predictable and controlled manner, and with hand-holding by a minimal and fixed number of programmers. Distributed sets of programs which are logically viewed as a single "application" should be configurable as a unit. This requires an unprecedented degree of cooperation between machines, and self-awareness by the machines.

One can explore this issue from two angles. The first is to explore the properties of a dynamically changing distributed application that runs on a variable set of machines. The other is to explore the properties of the operating systems running on the individual machines which will have to support these applications. In the Scout project, we are working a piece of the second problem. We do not claim a solution to these larger problems, but they frame our research.

3 Scout Overview

Scout descends from the x-kernel project[5], and is targeted towards creating kernels for network appliances, machines which have the property that while there is huge variation in the possible tasks they might perform, any incarnation is very specialized. Like the x-kernel, Scout kernels are built by composing modules, such as networking protocols, which together determine the nature of the system. Unlike the x-kernel, these modules are called *routers*, as they route the flow of data through the system. In addition to composition, Scout has the *path* abstraction: a flow of information originating in one router and terminating in another.

With routers and paths, Scout expresses its two fundamental operations: composition and per-connection specialization; the programmer uses these operations to specify the form of the kernel. Scout may be thought of a system for generating kernels, rather than a kernel itself. For problems easily expressed in these terms, such as network subsystems, it works very well. Outside of this realm things become more strained, but that is a topic of research for others.

My research is focused on pushing our module and path abstractions in three directions: making it possible to dynamically create and glue modules and paths together according to the capabilities and requirements of the kernel's environment; extending this flexibility and cooperation across multiple machines; and making this process as automatic, controllable, and efficient as possible.

3.1 Routers and Paths in Scout 1.0

It is beyond the scope of this document to fully describe the structure of Scout. For that, the reader is referred to the Scout Programmer's Manual[7].

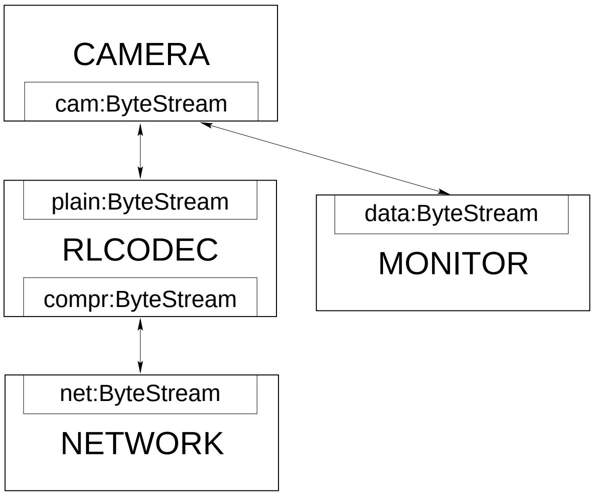

Figure 1: Router Graph for Network-Attached Camera

Routers are the fundamental unit of program development in Scout. Each Scout router is a module that implements a well defined and independent function. Well defined means that there usually is either a standard, interface specification, or other existing practice that defines the exact functionality of the router. Independent means that each router should provide a useful, self-contained service; i.e., the router should not depend on there being other specific routers connected to it. Typical examples of Scout routers include network protocols, such as IP, UDP, and TCP; storage system components, such as VFS, UFS, or SCSI; and drivers for the various devices in the system.

A Scout router exports a set of *services*, each of which provides access to one aspect of the router's functionality. For example, routers often provide filter-like functionality, which could be realized by a pair of services: one service representing the filter's input, the other representing the filter's output. The number of services provided by a given router is not limited to two, but is determined by the router.

Each service in a router has a *name* and a *type*. The names are unique, but otherwise arbitrary and chosen by the programmer. The type of a service is used by the Scout configuration editor to ensure that only mutually compatible services are connected.

To form a complete system, individual routers are connected into a *router graph*. For a trivial example, to build a network-attached camera whose video data should be run-length encoded before being sent out on the network, one could use the router graph shown in Figure 1. In this graph, `RLCODEC` represents the run-length codec introduced above, `CAMERA` is the device driver for the camera, `MONITOR` is a device driver for the small monitoring display of the camera, and `NETWORK` is the router implementing network support (in reality, the networking subsystem is likely to consist of multiple routers). The `RLCODEC` router has two services: *plain* and *compr*, both of type *ByteStream*.

The router graph for a scout system is currently specified at kernel link-time in a graph description file `config.graph`; it functions to specify which modules need to be linked into the executable, and constrains which paths may be created.

When one router wishes to open a connection to or through another router, it must do so by creating a path. The best way to think of a path is as the result of specializing the code in the kernel based on the invariants described by those attributes. Once this specialized path is created, then any data entering one endpoint of the path will be propagated along the path free from demultiplexing. In addition to providing a vehicle for various path-global optimizations, paths are the basic unit for all scheduling and resource allocation decisions.

A path is created with the function *pathCreate*() is called. This function takes as arguments the router which will start the creation, and an initial set of *path attributes* describing the new path. The first router will consult these attributes and create the first *stage* of the path. The stage is the data structure containing the data and functions needed for that router to perform its functionality for the new path. Then this router determines the next router in the path, and path creation continues there. This process continues until one of the routers is unable to make a unique routing decision, and so cannot determine the next router in the path. At this point, the set of stages is linked together to form the path data structure, and the path is established.

The graph specified in `config.graph` greatly simplifies the second task by limiting the possible neighbor routers. This job has also been unfortunately overly simplified by routers being hard-coded to only work with certain neighbors. What used to be the primary mechanism for composition in the x-kernel, the router graph, is really just an optimization to constrain the geometry of possible paths.

3.2 Impact of the Current Scout Design

The router and path primitives are to some degree in opposition. Routers are by definition semi-autonomous code modules, which are best written with minimal knowledge or assumptions about the inner workings of other modules. Paths are created based partly on "global" information, and span multiple routers. Therefore, when a path is created, routers interpret and affect the global invariants which define the path. This often requires routers to have knowledge and assumptions about things outside of their module, and strains the system's modularity. So, the first key problem is that routers contain inappropriate knowledge of global or application-specific state. An effect of this is that it can be very difficult to create paths not envisioned by the original router programmers. Also, this dispersal of global state among various routers greatly hampers the kernel's ability to control, configure, and monitor itself.

In addition to this information being inappropriately fixed in space, it is also inappropriately fixed in time. Some state, such as the router graph, can only be set at compile time, removing some run-time flexibility from the system. Other state, such as the total number of runnable threads, is only settable while the kernel is running, limiting the amount of static optimization or configuration possible. Likewise, the resulting Scout kernel may be too flexible. For example, it is currently difficult to set global policy about the nature and number of created paths.

In summary, Scout has two powerful, orthogonal operations: composition and specialization. Together they could be used to build up a wide variety of specialized network appliances, but Scout's current lack of dynamic control and self-awareness limits its effectiveness in spanning this space.

4 The Configuration Pipeline

Since Scout has less flexibility than it could, an obvious conclusion is to rewrite it so that it's more flexible. This is tricky however, since one of the major benefits of Scout is its ability to make specialized, and therefore inherently inflexible, kernels.

It is not sufficient, or even always wise, to make software which is of the utmost flexibility or configurability. In any engineering task, the end product benefits from being designed for a specific, well-defined role. It is no more desirable to make software which can be overly extended than it is to make software which is not extendable enough. Our goal is not to make software which is the most flexible, but to enable software which is exactly as flexible as needed. As this need can change, it is valuable if the programmer can dial in the degree and nature of configurability, just as he can specify which paths are to be created or which routers are included.

The *Configuration Pipeline* is an old, but useful abstraction. Much like compilers or rendering engines, the process of engineering software can be described as a pipeline of logically separate, but dependent, stages. The form of these stages depends on the particular application or engineering model, but they can generally be classified into two categories: static and dynamic. Static configuration is done before the executable image of the application is created. Dynamic configuration is done after the application has started running, usually in response to a changing environment or demands.

For example, consider a Scout kernel designed to run a server in a distributed video application. The interesting stages of its configuration pipeline has this general form:

1. Static configuration:

   Design At this stage the designers construct the abstract design of the system. We decided that Scout would have paths, and focus on networking protocols, and provide certain types of resource allocation.

   Programming While coupled with the design, it is not equivalent. The configuration choices made at this level determine the organization and form of the source code. Most programming choices occur at this level, such as where to obtain the source code for various device drivers, how to organize the schedulers, and which video types to support.

   Building Compiling and linking performs the ultimate specialization of the source code to run on a specific architecture. Also, partitions described by the source code between modules are realized in the generated code, which may be used later to provide security or dynamic loading.

2. Dynamic configuration:

   Initialization At this time, the kernel seeks out its invariant constraints, such as the hardware configuration or the available scheduling algorithms, and performs the necessary initialization.

   Connection Setup Various actions must be performed when a request is made to this machine to start serving a video, such as resource allocation.

   Connected There are myriad interactions inside the kernel in order to adapt the compression of the video to suit the bandwidth available in the network.

It is immediately apparent that the static stages are more clearly classifiable and ubiquitous than the dynamic stages. There are two reasons: they have been researched to a greater degree, and the dynamic stages are more dependent on the function and design of the software.

A primary goal of our research is to determine the best way to coordinate the stages of this pipeline in the Scout system. Pushing configuration decisions towards the front of the pipeline offers opportunities for optimization. Pushing decisions towards the back offers runtime flexibility. For example, if it is known at compile time exactly which paths and threads will be running, it is possible to statically compute a optimal scheduling algorithm. Likewise, when compilation can be performed at runtime, it becomes convenient to move code between machines. If the decisions dependent on the hardware devices in the computer can only be made at boot time, then the machine must be rebooted when a new device is added.

The key is the recognition that decisions are movable between stages, and the best results are obtained if the configuration properties of the system are well understood and chosen for the task at hand. Depending on the constraints of the programmers, the hardware, and the intended environment of the machine, different decisions have different optimal points in the configuration pipeline. Some decisions are best made at connection setup. Some decisions are best handled at the design level. A relatively small change in the environment, such as the development of PCMCIA, can trigger a large-scale migration of these decisions. It is best to be prepared for this, and to be able to react as efficiently as possible.

5 Ranger

Ranger is a new subsystem to Scout (actually implemented as a router), which attempts to perform and control path creation and other forms of kernel configuration. Basically, information which is logically local to a given router should only exist in that router. Information which is shared between routers, or which does not logically belong to any, should be controlled by a third party. One of Ranger's jobs is to manage this global information necessary for path creation and kernel configuration. This requires that all routers agree on a clear definition of this information. The other job of Ranger is to coordinate the routers' efforts to knit together a path, rather than the old method which required the routers to blindly negotiate with each other - a process which can often only create paths which were envisioned by the kernel designers. In addition to this information coming from different places in the kernel, this information can be generated at different times. Some invariants are known at design time. Others only at connection setup time. A key goal in designing Ranger is keeping *when* a piece of information is generated or used independent from *where* it is generated or used.

While Ranger will increase the range of Scout kernels which can be created, the main benefit is that the kernels can be configured and controlled much more efficiently, whether by programmers, users, or the kernel itself. It will bring Scout a large step closer to the goal of being a *breadboard* OS, which a single programmer (or a software agent such as active network code) can use to quickly plug together a functional and efficient kernel.

Ranger consists of four components:

1. A database of information necessary for path creation and kernel configuration. This includes the various invariants required or recognized by routers, and the global state effecting path creation. It does not particularly matter when the information is entered into the database. Some will be set at compile time, particularly the router graph currently described in config.graph. Other will be set while the system is running, such as which routers are currently loaded.

2. The path creation interface between Ranger and the routers. For cleanliness, the topology discovery phase of path creation will be done before the routers create their stages. This allows for separating the code which initializes the stage (which is often boilerplate and can be generated), from the more involved code which determines the topology of the path. It also allows the discovery phase to occur in a different order through routers than the establishment phase, allowing for such things as creating a path from the endpoints in, or a midpoint out, or backtracking.

3. Code for monitoring and controlling this path creation process. In particular, monitoring and controlling the topology discovery process. This way we can efficiently blend router-local information and requirements with the global state of the system. For example, the TCP router may only care that it needs to talk to the network device through an IPv4 router, but Ranger may know which particular IPv4 router it can, or should use. Ranger will have the facility to interact with other parts of Scout during this process - for example a security manager most likely would want to control which routers can be neighbors.

4. An interface for interacting with other parties interesting in controlling or monitoring Ranger's activity. Such parties may be applications which require the coordinated creation of a set of paths, kernel subsystems responsible for keeping order (such as the Escort[8] security manager), the resource allocation modules, or active code downloaded into the system to perform remote monitoring or configuration of the running system.

It is an interesting point that Ranger is itself a router. Any router can use the path creation API to create paths. It just happens that Ranger is particularly good at it and will be used for the most interesting paths, while the short, simple, easily defined paths needed for bootstrapping the system without it.

The Ranger path creation algorithm is straightforward to summarize. The path attributes are divided into two classes: those which affect the topology of the path, and those which just influence state within one or more stages. The topology attributes are primarily needed during the route discovery phase, although they are useful during path establishment as well. The state attributes are only needed for establishment.

These path attributes have the form:

$$\{TargetRouter, Invariant\}$$

The *TargetRouter* is a value which can be resolved to a candidate router. For example, it can range from being the specific name of a router, all the way the set of whichever routers is connected to a router's service interface. The *Invariant* is just an invariant which the *TargetRouter* must satisfy. For topology attributes, the *TargetRouter* field provides the mechanism for specifying the directionality of an invariant. For example, `TCP` only requires `IP` in one direction. An `IP` protocol splicer router will require `IP` in both directions. For state attributes, that field enables signalling a specific router or set of routers.

Ranger initiates path creation when it is given a set of initial attributes. It can then modify this set depending on the internal database, and determines the initial *anchor* router or routers which must be in the path. For example, if given a vague attribute requiring some version of IP, it can replace this with an attribute resolving to a specific IP router. Likewise, if given a specific attribute which resolves to a router which the client is not allowed to access, then it may reject the path creation request.

Ranger then queries the anchor router and depending on what that router requires, develops a new set of attributes. Ranger then iterates until either a path can be created with no unsatisfied attributes, or else returns an error to the client.

6 Conclusion

Creating software which is appropriately flexible and configurable is always difficult. When successfully done, the benefits are powerful. Bloated and possibly useless software results from failure. Emacs is often cited as an example (of which is a contentious topic). Regardless, the increasing power of computers, numbers of programmers, and appetites of users is only increasing the desire to create such software.

Since Scout is a system for generating varied and specialized kernels, it is a useful platform for exploring this problem. Scout can be used to generate a kernel with a specified set of protocols, or preset resource allocation strategies. Flexibility can be considered a resource. As such, it should not be artificially limited without just cause; the programmers and users should have as much control over it as feasible. The key parts of Scout, such as the network system, consists of modules which have well-defined and understood behavior and composability; this can be used to achieve a level of automation difficult to achieve in other types of software. Ranger already shows great promise as a partner system for the Escort security manager, and may well prove a useful tool for kernel configuration.

References

[1] Brian N. Bershad et al. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[2] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: A system for constructing fine-grain configurable communicationservices. Technical Report TR97-12, University of Arizona, Department of Computer Science, July 7, 1997.

[3] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specific resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, December 1995.

[4] A. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. In *Commun. ACM*, volume 19, no. 5, pages 266–272, May 1976.

[5] Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[6] David Mosberger and Larry Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 153–168, October 1996.

[7] Scout programmer's manual: http://www.cs.arizona.edu/scout/manual/.

[8] Oliver Spatscheck and Larry Peterson. Escort: A path-based OS security architecture. Technical Report TR97-17, The Department of Computer Science, University of Arizona, Monday, November 17 1997.

[9] D.L. Tennenhouse and D. Wetherall. Towards an active network architecture. In *Multimedia Computing and Networking 96*, 1996.

[10] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in horus. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 80–89, Ottawa, Ontario, Canada, 2–23 August 1995.

[11] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *CACM*, 39(4):76–83, April 1996.

[12] David Wetherall, John Guttag, and David Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH 98*, San Francisco, CA, April 1998.