Distributed Application Development with Inferno

Ravi Sharma
Inferno Network Software Solutions
Bell Laboratories, Lucent Technologies
Suite 400, 2 Paragon Way
Freehold, NJ 07728
+1 732 577-2705
sharma@lucent.com

ABSTRACT

Distributed computing has taken a new importance in order to meet the requirements of users demanding information “anytime, anywhere.” Inferno facilitates the creation and support of distributed services in the new and emerging world of network environments. These environments include a world of varied terminals, network hardware, and protocols. The Namespace is a critical Inferno concept that enables the participants in this network environment to deliver resources to meet the many needs of diverse users.

This paper discusses the elements of the Namespace technology. Its simple programming model and network transparency is demonstrated through the design of an application that can have components in several different nodes in a network. The simplicity and flexibility of the solution is highlighted.

Keywords

Inferno, InfernoSpaces, distributed applications, Styx, networking protocols.

1. INTRODUCTION

The growth in the Internet has led to an information explosion with users demanding information, "anytime, anywhere". Today, applications, services and information are distributed in and accessed from multiple physical locations. To meet the "anytime, anywhere" requirements, the focus on developing computing technologies to universally access, manage and present services and information has intensified. Distributed computing has taken a new importance in this context and most of the problems (reliability, recovery and concurrency, to name a few) associated with these requirements are addressed in this domain. As a result, there are numerous distributed computing paradigms (object and file based) and solutions available to an application developer.

Inferno facilitates the creation and support of distributed services in the new and emerging world of network environments. These environments include a world of diverse terminals, network hardware, and protocols. Inferno is designed to insulate the diverse providers of content and services from the equally varied transport and presentation platforms. A critical Inferno concept enables the participants in this network environment to present their resources as files in a hierarchical name space. The objects appearing as files may represent stored data, but may also be devices, dynamic information sources, interfaces to services, control points. This approach unifies and provides basic naming and structuring mechanisms for the system resources. The approach is usable even without adopting a new operating system.

The **Inferno** operating system provides a software infrastructure for distributed, network applications that allows any application, written in the Limbo programming language, to run across multiple platforms and networks under the Dis virtual machine. Inferno provides an elegant file-like interface to resources and services that allows the dynamic construction of a user Namespace (described in the next section). An Inferno application can access the resources and services in its Namespace even though they may be distributed throughout the network.

InfernoSpaces extends many of the Inferno Namespace capabilities to non-Inferno platforms. It is currently available as a software component in C and as a class library in Java on Solaris and Windows operating Systems. InfernoSpaces can be extended to support additional languages, operating systems and hardware platforms. InfernoSpaces allows legacy applications to easily take advantage of the Inferno capabilities.

2. NAMESPACE

The Inferno Namespace operations provide a powerful set of features for delivering distributed applications. They provide a simple programming model and network transparency while also providing great flexibility in delivering resources to meet the many (sometimes conflicting) needs of diverse users. It results in applications that are simpler to develop, have fewer lines of code, are scalable and are easier to maintain than those on other platforms.

The Inferno *Namespace* is the hierarchy of resources available to a program. There are two features that make Namespace a major component of the Inferno system. First, Inferno represents most resources as files. Files are not just data; files may also be devices, network connections, and interfaces to services. So, the (file) Namespace actually represents a diverse resource space. Second, Inferno and InfernoSpaces offer a unique set of operations to manage that Namespace. These operations allow programmers and application developers to cope with complexity in today's networked and distributed environments.

Even within a single system, this hierarchical structure with its attached access control provides a familiar scheme for naming, classifying, and acquiring the system resources. More important, this approach provides a very natural way to build distributed systems, because the technology for attaching remote file systems is well known. In brief: if system resources are represented as files, and there are remote file systems, you have automatically constructed a distributed system, just because the resources available in one place are usable from another place.

3. INFERNO DESIGN

Inferno was designed with a technical model of three basic principles. First, all resources are named and accessed like files in a forest of hierarchical file systems. Second, the disjointed resource hierarchies provided by different services are joined together into a single, private name space. Third, a communication protocol, Styx, is applied uniformly to access these resources, whether local or remote. Applications see a fixed set of files organized as a directory tree. Some of these files contain ordinary data, but others represent more active resources. System services live behind file names. Devices themselves are also represented as files, and device drivers attached to a particular hardware system present themselves as small directories. These directories typically contain two files, data and ctl, which respectively perform actual device I/O and control/status operations.

The glue that connects the separate parts of the resource name space together is the Styx protocol. Within an instance of Inferno, all the device drivers and other internal resources respond to the procedural version of Styx. The Inferno kernel implements a mount driver that transforms file system operations into remote procedure calls for transport over a network. On the other side of the connection, a server unwraps the Styx messages and implements them using the local view of resources (which themselves may have been mounted from other remote locations). The Styx protocol lies above and is independent of the communications transport layer; it is readily carried over TCP/IP, PPP, etc.

This approach has a number of advantages:

- Simple programming model
- Small footprint (~2K lines of C for the interface code)
- Platform and language independence
- Built-in hierarchy fits most system design
- Component-wise debugging of distributed systems and dynamic reconfiguration
- Small, simple, precise definition

Inferno creates its own standard environment for applications. Applications are written to execute in a virtual machine to provide true portability and efficient execution on a variety of native hardware platforms. Identical application programs can run under any instance of this environment, even in distributed fashion, and see the same resources and kernel services. These kernel services include process management and scheduling, memory management and garbage collection, namespace construction and sharing, and device driver event management.

The purpose of most Inferno application is to present information or media to the user; thus applications must locate the information sources in the network and construct a local representation of them. The information flow is not one-way; the user's terminal is also an information source, and its devices represent resources to applications.

In practice, most applications see a fixed set of files organized as a directory tree. Some of the files contain ordinary data, but others represent more active resources. Devices are represented as files, and device drivers attached to particular hardware present themselves in small directories for status, control, and data access. System services also live behind file names.

Inferno creates a standard environment for applications. Identical application programs can run under any instance of this environment, even in distributed fashion, and see the same resources. Depending on the environment in which Inferno itself runs, there are several versions of the Inferno kernel, interpreter, and device-driver set.

When running as the native operating system, the kernel includes all the low-level glue (interrupt handlers, graphics, and other device drivers) needed to implement the abstractions presented to applications. For a hosted system, for example under Unix, Windows NT, or Windows 95, Inferno runs as a set of ordinary processes and adapts to the resources provided by the underlying operating system.

4. THE STYX PROTOCOL

Styx is the native file access protocol of the Inferno operating system. It provides a view of a hierarchical, tree-shaped file system name space, together with access information about the files (permissions, sizes, and dates) and the means to read and write the files. Its users (that is, the people who write application programs), don't see the protocol itself; instead they simply see files that they read and write, and that provide information or change information.

In use, a Styx *client* is an entity on one machine that establishes communication with another entity, the *server*, on the same or another machine. The client mechanisms may be built into the operating system, as they are in Inferno, or into application libraries such as InfernoSpaces; the server may be part of the operating system, or just as often may be application code on the server machine. In any case the client and server entities communicate by exchanging messages, and the effect is that the client sees a hierarchical file system that exists on the server. The Styx protocol is the specification of the messages that are exchanged.

At one level, Styx consists of messages of 13 types for:

- Starting communication (attaching to a file system)
- Navigating the file system (that is, specifying and gaining a handle for a named file)
- Reading and writing a file
- Performing file status inquiries and changes

However, application writers simply write requests to open, read, or write files; a library or the operating system translates the requests into the necessary byte sequences transmitted over a communication channel. The Styx protocol proper specifies the interpretation of these byte sequences. It fits, approximately, at the OSI Session Layer level of the ISO standard classification. Its specification is independent of most details of machine architecture and it has been successfully used among machines of varying instruction sets and data layout.

At a lower level, implementations of Styx depend only on a reliable, byte-stream Transport communications layer. For example, it runs either over TCP/IP or over IL, which is a sequenced, reliable datagram protocol using IP packets.

4.1 Architectural approach

Styx, purely as a file system access protocol, is distinguished by its simplicity and coherence, but that in itself is not enough to urge its adoption; instead, it is a component in a more encompassing approach to system design: the presentation of resources as files. As an example, access to a TCP/IP network in Inferno systems appears as a piece of a file system, with (abbreviated) structure as follows:

```
/net
        /dns
        /tcp
                clone
                stats
                /0
                        /ctl
                        /status
                        /data
                        /listen
                /1
                        ...
                ...
        /ether0
                /0
                        /ctl
                        /data
                        ...
                /1
                        ...
                ...
```

This represents a file system structure, in which one can name, read, and write 'files' with names like `/net/dns`, `/net/tcp/clone`, `/net/tcp/0/ctl` and so on; there are directories of files `/tcp` and `/net/ether0`. On the machine that actually has the network interface, all of these things that look like files are constructed by the kernel drivers that maintain the TCP/IP stack; they aren't real files on a disk. Operations on the 'files' turn into operations sent to the device drivers.

Suppose an application wishes to establish a connection over TCP/IP to `www.bell-labs.com`. The first thing it must do is to translate the domain name `www.bell-labs.com` to a numerical internet address; this is a complicated process, generally involving communicating with local and remote Domain Name Servers. In this model, the action required is to open the file `/net/dns` and write the literal string `www.bell-labs.com` on the file, and then to read from the same file, receiving the string `204.178.16.5`.

Once the numerical Internet address is acquired, the connection must be established; this is done by opening `/net/tcp/clone` and reading from it a string that specifies a directory like `/net/tcp/43`, which represents a new, unique TCP/IP channel. To establish the connection, write a message like `connect 204.178.16.5` on the control file for that connection, `/net/tcp/43/ctl`. Subsequently, communication with `www.bell-labs.com` is done by reading and writing on the file `/net/tcp/43/data`.

There are several things to note about this approach:

- All the interface points look like files, and are accessed by the same I/O mechanisms already available in programming languages like C, C++, or Java. However, they do not correspond to ordinary data files on disk, but instead are creations of a middleware code layer.
- Communication across the interface, by convention, uses printable character strings where feasible instead of binary information. This means that the syntax of communication does not depend on CPU architecture or language details.
- Because the interface, as in this example with `/net` as the interface with networking facilities, looks like a piece of a hierarchical file system, it can easily and nearly automatically be exported to a remote machine and used from afar.

5. THE CHAT APPLICATION

The chat application is an example of distributed computing, implemented using the Namespace concept. This example consists of a chat server and several chat clients. The chat server supports multiple chat sessions, maintains a list of chat messages for each chat session, and provides a dynamic list of participants for all chat sessions. The chat clients, on the other hand, can request a list of active chat sessions and participants from the chat server, create, join or leave a chat session. Once in a session, chat clients can send and receive messages to and from the session.

In this example, the Namespace concept is implemented on an emulated version of Inferno. A similar chat example has been implemented using InfernoSpaces. As long as the chat server and clients use the Namespace technology (either Inferno or InfernoSpaces), they will interoperate.

A Namespace is created to represent a chat server. The set of processes and functions that represent the chat server functionality are modeled as a set of synthetic files. For the chat application, these are operated on by simple file operations (read and write). Since a Namespace represents the chat server, the synthetic files can be resident anywhere on the network. Taking this one step further, some of the synthetic files that constitute the chat server Namespace could be in one part of the network while others can be elsewhere in the network, i.e., the functionality that constitute the chat server could be physically distributed in the network.

The chat server consists of two synthetic files, `chat` and `chatctl` in the `/chan` directory (see figure 1). The `chat` file represents a chat session. To send messages to other participants in a chat session, the chat client writes a message to the `chat` file. To receive messages from participants it reads from the `chat` file. The `chatctl` file is used for status information. To find out what chat sessions exist is simple as performing a read on the `chatctl` file.

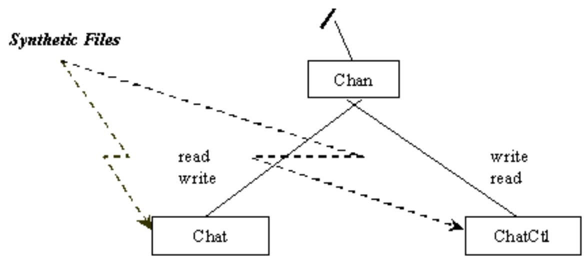

Figure 1. Chat Server Application Namespace

A chat client needs to access the server Namespace for a chat session. To do so, the chat client mounts the `/chan` directory on the chat client node, `/mntpt`, in this example (see figure 2). It is now able to access, `chat` and `chatctl` and begin chatting.

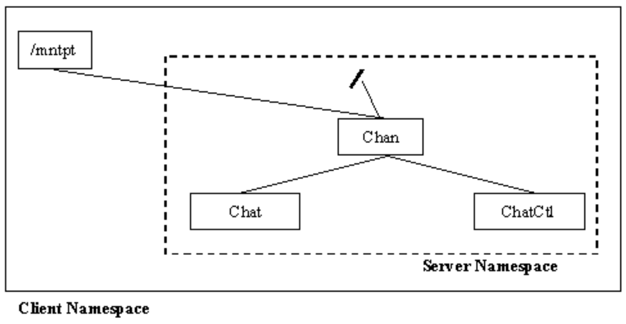

Figure 2. Client and Server Namespace

The chat application was implemented in three steps:

- Decompose functions into a set of synthetic files
- Create the Namespace representing the chat server
- Mount the synthetic files on the chat client

Once the Namespaces of the servers and clients are created, it is only a matter of manipulating the simple file operations, read and write, that are familiar to most programmers, for the Chat application to work.

6. INFERNO AS AN OS

Inferno is a small full-service operating system. Complete kernels including basic applications are available in 1MB of memory. The architecture of the Inferno kernel is depicted in figure 3. Inferno's capabilities include its own file system, threads, networks, and other basic services.

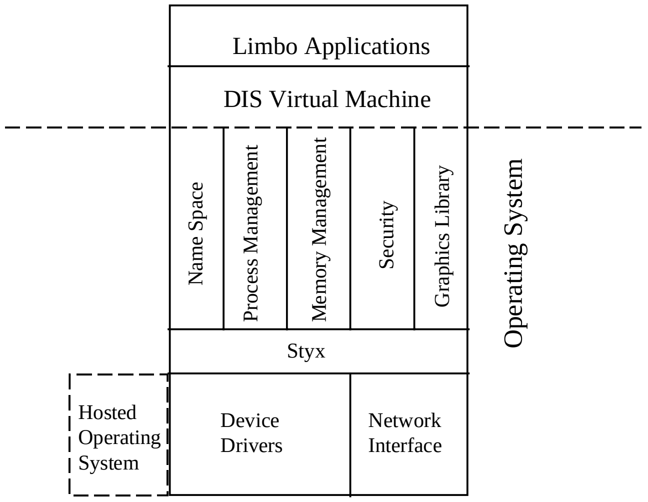

Figure 3. Inferno OS Architecture

Porting Inferno to a new platform is a relatively straightforward task. In general, the kernel code consists of a portable part shared by all architectures and a processor-specific portion for each supported architecture. The portable code is often compiled and stored in a library associated with each architecture. The kernel is built by compiling the architecture-specific code and loading it with the libraries containing the portable code. Support for a new architecture is provided by acquiring or building a C compiler for the architecture, using it to compile the portable code into libraries, writing the architecture-specific code, and then loading that code with the libraries.

The architecture-specific code is characterized by several simple functions that need to be implemented in an architecture-specific way. An atomic test-and-set function, interrupt-level management functions, scheduler label information (program counter and stack-frame pointer) management functions, floating-point unit functions, etc., are all examples of such architecture-specific code.

The new port will also require device drivers appropriate for its configuration. These drivers can be acquired from currently available drivers, adapted from an existing driver, or implemented from scratch. The kernel will also need to be loaded onto the target platform; this loading is typically provided for by a platform-specific boot loader that also provides memory unit initialization and interrupt-level initialization functions. The kernel initialization completes the startup required for the platform, by initializing the drivers loaded into it.

7. PERFORMANCE CONSIDERATIONS

In any distributed application environment, several tradeoffs must be considered to provide effective service. Within the kernel, the amount of message queuing, mutual exclusion locking, and event management by various kernel processes and device drivers affords numerous opportunities for optimization. The distribution of an application across multiple computing nodes exposes any inefficiencies of communication across the partitions of the application. Not only does this type of distribution demonstrate a classic confrontation of communication bandwidth versus processing power or computational resource, service delivery to numerous end-points brings into question additional issues of reliability, scalability, administration, and security. Dynamic use of Inferno Namespace mechanisms offers simple yet innovative solutions to either side of engineering exercise.

8. SUMMARY

Inferno facilitates the creation and support of distributed services in the new and emerging world of network environments. These environments include a world of diverse terminals, network hardware, and protocols. Inferno is designed to insulate the diverse providers of content and services from the equally varied transport and presentation platforms.

Inferno and InfernoSpaces implement the Namespace technology. The Inferno Namespace operations provide a powerful set of features for delivering distributed applications. They provide a simple programming model and network transparency while also providing great flexibility in delivering resources to meet the many needs of diverse users. It results in applications that are simpler to develop, have fewer lines of code, are scalable and are easier to maintain than those on other platforms are. Though they are ideally suited to resource and memory constrained devices there is no restriction on the size of the platforms that they can be used in. InfernoSpaces brings the Namespace technology to non-Inferno platforms. Applications developed in popular programming languages (C, C++, Java, …) can take advantage of the Namespace technology with InfernoSpaces and will be able to inter-operate with other InfernoSpaces and Inferno platforms.

9. ADDITIONAL REFERENCES

[1] Inferno Home Page. http://www.lucent.com/inferno.

[2] Dorward, Sean M., *et al*, "The Inferno Operating System", *Bell Labs Technical Journal*, Volume 2, Number 1 (Winter 1997), pp. 5-18.

[3] Mooken, Thomas, "Inferno, InfernoSpaces, and Distributed Computing", *Proceedings of the Embedded Systems Conference*, Spring 1999, Chicago, IL.

[4] Rau, Larry, "Inferno: One Hot OS", BYTE, Volume 22, Issue 6 (June 1997), pp. 53-54.

[5] Sharma, Ravi, "Inferno, Limbo take Java to coding task," *EE Times*, January 1, 1997, p.60.