System Overview 5

Introduction

This chapter provides an overview of the Inferno system, including:

- Innovative Basic Principles
- Design Characteristics
- System Architecture
- System Components

Innovative Basic Principles

The Inferno system is a distributed operating system based on three innovative principles:

- **Resources as files:** System resources are represented as files in an hierarchical file system.
- **Namespace:** The application view of the network is a single, coherent namespace that appears as an hierarchical file system but can represent physically separated resources.
- **Standard communications protocol:** A standard communications protocol, called Styx, is used to access all resources, both local and remote.

Resources as Files

File systems are central to the Inferno system. Files are not just data, but also interfaces to peripherals, programs, services, and networks. This file-oriented interface involves resolving names in an hierarchical tree, attaching to them by name, and accessing their contents by *read* and *write* calls.

Using files as a central concept in the system has the following advantages:

- File systems have simple and well understood interfaces across a wide variety of operating systems. Interfaces to the files generally consist of a small set of well-defined operations, such as *open*, *read* and *write*.

- Reliance on file systems reduces the amount of interface code and keeps the Inferno system small, reliable and highly portable. The use of a file system interface also reduces the complexity and size of applications.
- Naming conventions for files are well known, uniform, and easily understood.
- Access rights and permissions to files are simple, yet can be used to ensure multiple levels of security.

File names and contents can be dynamic and can be synthesized on per-demand and per-client basis. For example, the Inferno system creates a dynamic set of files to represent each active TCP connection. An application then reads from and writes to these files while the target TCP connection is still active. When the connection is no longer necessary, the files are closed and the connection is closed automatically.

The Inferno file system structure is defined implicitly by fixing the allowable set of file methods. File servers can easily manage the brokerage of file operations such as granting file access. The file system structure implicitly defines the mechanisms for garbage collection and accounting.

Namespace

The second key principle of the Inferno system is the customizable namespace by which an application builds a unique private view of the resources and services that it needs to access. Each set of resources and services is represented as an hierarchy of files and is accessible using familiar file access operations of *open*, *read*, and *write*. The various resources and services used by a process are combined into a single hierarchy of file names called a *namespace*.

The resources accessible to an individual namespace can be located on a single client or on multiple servers throughout the network. Therefore, file systems provided by different servers are combined into a single, unique namespace, which becomes the application's view of the network.

The Inferno namespace operations provide a simple programming model, network transparency, and great flexibility in delivering resources to meet the many, and sometimes conflicting, needs of diverse users. The use of namespace and namespace operations results in simpler, smaller, less costly, and more maintainable software.

Reduces Complexity

The file model is simple (*open*, *read*, and *write*) and known to all programmers. Representing so many resources by a common interface is a first step in controlling complexity. For certain naturally complex resources, the Inferno system provides interfaces as suites of related files, each individually simple to use. One example is the Internet Protocol (IP) device interface that includes separate data, ctl (control), and status files. Devices are controlled by simply writing a text string to the appropriate control file. For example, a connection can be established by writing "connect 135.3.60.100" to the control file.

The local namespace can be extended by grafting on file trees from Inferno servers, thereby gaining access to resources from those machines. Once incorporated into the local namespace, use of the remote resources is indistinguishable from local resource use.

Namespace operations can be used to separate application issues from configuration issues. Resources can be made to appear where they are needed rather than having applications adapt to where resources can be found.

The namespace allows for network transparency that is supported for all Inferno files including files of data, devices, and daemon interfaces. Applications are networked without explicit network programming.

Reduces Size

Some system interfaces that traditionally require system calls are provided by special files. This reduces the size of the Inferno operating system.

Customizable

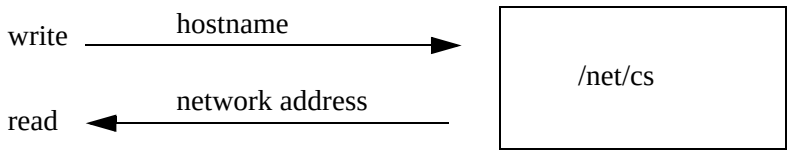
The namespace can be personalized. Changes in the namespace can be either shared or restricted to individual users or even to individual programs. In this way, resources can be optimally positioned within the namespace for each situation rather than having to find compromises between the varied, sometimes contradictory, needs of many applications.

Connectivity

An important system service provided through a file interface is the connection server, a service that translates a symbolic host name to a network address, which can be used to establish a connection.

The protocol is to write the hostname of interest into the special file */net/cs*, then read back the answer.

Figure 5-1 Connection Server



The write request is detected by a daemon process that translates the name and stores the answer until the read request arrives. Multiple, interleaved requests are handled correctly. The daemon is a small, simple application program, not part of the kernel. The kernel itself uses the */net/cs* file to translate a name to a network address.

Namespace Operations

The key namespace operations are *mount* and *bind*. See the section on *Namespace Construction* later in this chapter.

Standard Communications Protocol

The third key principle of the Inferno system is the use of a standard communications protocol called *Styx*. This protocol represents network transports, network devices and network connections as file systems.

The protocol supports remote access to files and allows a remote machine to use these interfaces as gateways. Using the same set of files to represent different devices allows the creation of common tools to serve several networks. Network connections represented by these files behave the same way for all networks and this allows applications to contain no network-specific code.

Design Characteristics

The characteristics that result from the basic design of the Inferno system are:

- Small
- Portable
- Secure
- Network independent

Small

Since resources are represented as files, the amount of interface code required for the kernel and the applications is significantly reduced. Complete solutions, including basic applications, can run comfortably in one megabyte of memory in an Inferno-enabled network.

Portable

The Inferno kernel and device drivers, the emulation kernel, and the Dis interpreter are written in C to provide a high degree of operating system portability. The remainder of the Inferno system is written in Limbo, a high-level language designed to take advantage of Inferno system capabilities. Writing these components in Limbo ensures that they will behave the same way on any platform to which the Inferno system is ported. In addition, application programs written in Limbo and compiled to Dis bytecode will behave the same way on any Inferno system platform.

Note: Device drivers for specific hardware, such as an MPEG card or television interface, must be implemented when the Inferno kernel is ported to a new hardware environment.

Secure

The Inferno system authenticates all connections between systems with *digital signatures* using the ElGamal algorithm. This authentication uses the public key algorithm, in which both client and server possess public key/private key pairs. Messages signed with a private key can be verified with the public key.

In addition to authentication, the Inferno Application Programming Interface (API) also provides *message digesting* using the SHA or MD5 algorithms and *data encryption* using RC4 and DES.

Security at the application level is the responsibility of the application designer. The Inferno API provides a generic way to implement security in all Limbo applications.

Note: Security algorithms are subject to US Department of Commerce export restrictions. Not all security components described in this guide are included with the distribution.

Network Independent

Use of a standard communications protocol allows Inferno applications to be designed independently of the physical network. It is not necessary to design and develop local and remote proxy interfaces to ensure inter-machine communications.

System Components

This section provides an overview of the components of the Inferno system.

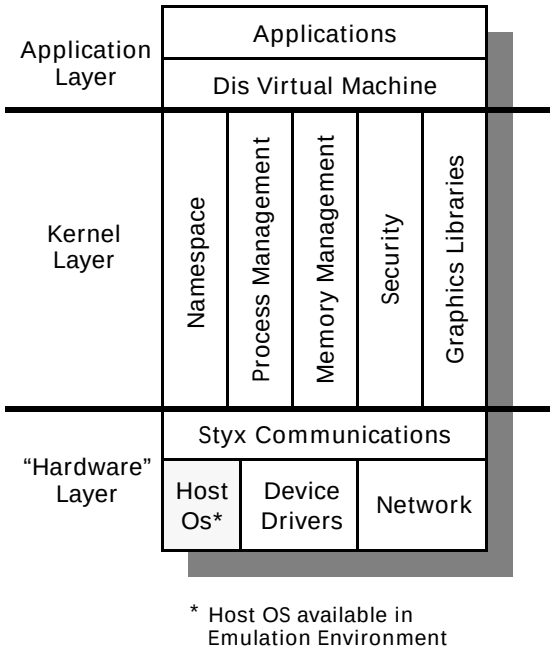
The four primary components of the Inferno system are:

- Inferno kernel
- Limbo programming language
- Dis virtual machine
- Styx communications protocol

Each of these elements provide distinct functions, integrated together to make up the Inferno operating system.

Figure 5-2 presents the components of the Inferno system as layers.

Figure 5-2 Inferno Components Diagram



Application Layer

At the highest level, the application layer, Limbo applications run through the Dis virtual machine. The Dis virtual machine component of the application layer interprets application instructions and invokes the appropriate system calls necessary to complete the requested actions.

Applications written in Limbo can run on any Inferno resource, either a client or a server. The interface to the lower levels of the Inferno system, the Inferno API, is identical for both client and server, and across all Inferno environments. The Inferno API provides access to system

resources by including core Limbo modules that contain explicit interface specifications.

One of the major benefits of the Inferno system is that it insulates applications from the hardware, operating system, and the network. This helps reduce programming costs since a single program will run on multiple types of hardware, either native or alongside host operating systems, and with various types of network connections. You do not have to have multiple versions for multiple platforms.

Kernel Layer

Below the application layer are the components of the Inferno system that provide operating system functionality, including:

- Namespace management
- Process management
- Memory management
- Security
- Graphics functions

These kernel elements behave the same whether an Inferno kernel is running directly on hardware as the native operating system or as an emulator on a host operating system.

“Hardware” Layer

The hardware layer primarily consists of physical components of the system, such as devices and networks. The exception is when the Inferno system is running in the emulation environment, where access to this layer is through the host operating system. In either case, the application code is insulated from direct access to the hardware.

Kernel

The Inferno kernel provides mechanisms for data and resource management, networking, and security.

The purpose of the Inferno kernel is to provide an environment for running applications under the Dis virtual machine. The kernel provides the following services to Dis:

- Process management
- Memory management
- Namespace management
- Data streaming
- Network protocols

The kernel is also responsible for I/O and hardware connections through device drivers. The reduced role of the kernel results in an operating system that is small in size and, therefore, easier to port to new platforms and architectures.

Native versus Emulated Environments

The Inferno kernel runs in one of two modes:

Native: The Inferno system runs as the primary operating system, directly on the CPU.

Hosted: The Inferno system runs as an application on the native operating system. This is called emulation mode.

The goal of the emulation kernel is to hide the differences between operating systems. For purposes of application development, the emulation kernel is identical to the native kernel. Only the small proportion of hardware- or operating system-dependent code differs in each configuration. And, with the new Device Driver Development Kit (DDK) available with Release 2.0, writing a device driver is easier using the standard kernel interface.

In a hosted, or emulation, environment, the system is much the same as in native mode. In a hosted environment, a host operating system provides many of the services that the native Inferno kernel provides in the native environment. The Inferno system can efficiently perform resource management that might be dependent on the devices on the host system, allowing the Inferno system to interact with existing operating systems without modification to the upper, application layer.

In hosted mode, the Inferno emulator, called *emu*, provides the control console of the system. This console is in the form of a shell-like interface from which other Inferno applications can be launched. The virtual machine insulates applications from the host operating system and architecture. For example, system calls for memory management and scheduling algorithms are the same regardless of the platform. In hosted or emulated mode, the Inferno system is an application to the host operating system. The Inferno system makes use of the host drivers. The host is unaware of Inferno applications.

While many resources are available to the Inferno system when it is run in the hosted mode on a server with many resources, native mode is used to run the Inferno system on thin clients with small memory and no disk. The native Inferno system is compiled to and makes calls directly to the hardware. The kernel and Dis have to be modified. Inferno device drivers have to be written. When the Inferno system is booted, a program called **init** is executed. This program can invoke any Inferno application or shell.

Process Management and Scheduling

The Inferno kernel provides preemptive scheduling of processes that are responsible for managing and servicing protocol stacks, media copies, alarms, interrupts, and the like. The kernel schedules processes with multiple priority run queues using a round-robin mechanism. Scheduling is on a fixed time slice, or quantum, basis with each quantum being set by the local system clock.

The run queue has eight classes, ranging from background, the lowest priority, through lock, the highest priority. The highest priority is given to tasks such as video telephony followed in order of decreasing priority by MPEG codec, audio codec, important, normal, low and background tasks. Class switching is supported using the system module stream method. The stream process is moved to the highest priority to perform media copies.

Interrupts are fully preemptive with the exception of processes holding spin locks. The priority of these processes is raised to ensure task completion.

Memory Management

The Inferno system has a two-level memory allocation mechanism. The lower level maintains control of large blocks of contiguous memory. The higher level divides memory into pools to allow applications to control the behavior of the system when resources become scarce. The allocation of memory pools for various purposes as required by the operating system is provided for:

- The allocation of general memory structures
- Memory for the Dis heap
- Memory for graphics images and fonts

- A network buffer pool

Memory is stored as blocks in an unbalanced B-tree with the leaves sorted by size. When a process requires memory, the tree is searched, using a best-fit algorithm, for a block of memory that is large enough. If the available block of memory is more than 25% larger than required, the block is split. One piece of the block is allocated to the process and the remainder is returned to the tree. If a block is less than 25% larger than required, the block is allocated, wasting the remainder.

When the memory is released, the block is returned to the tree. If the block is placed adjacent to another empty block on the tree, the two are merged to form a single available block. This memory management policy generally provides for fast allocation and relatively low fragmentation of internal memory.

Devices

Each Inferno system device implements a file tree that can be attached to a namespace for access by applications. The interface to all devices is through the Inferno file system interface. Each device is represented by a set of files that is implemented by a device driver. Device-specific behavior is implemented by responding to *open*, *read* and *write* system calls. With Release 2.0 and the new Device Driver Development Kit (DDK), a device driver writer can use the standard kernel interface. The developer no longer needs to have detailed knowledge of the kernel to write a device driver. Inferno system installations in an emulation environment require file system interfaces and interface programs to connect to local device drivers and the file system implemented by the host operating system.

File names beginning with the pound sign (#) specify the root of a file tree that is implemented by a kernel device driver. The character following the pound sign identifies the particular device driver. Table 5-1 lists the

Inferno system device characters. Some characters are reserved and other drivers may be added in later releases.

Table 5-1 Inferno System Devices

/	Root file system, used to construct bootstrap namespaces
|	Pipe
A	Digital audio input /output, provides access to stereo audio ports
C	Command execution
c	Console device, controls console I/O, keyboard and mouse processing
D	Secure sockets layer, provides encryption support
d	Draw device, implements raster graphics on a remote display
E	MPEG device, controls streaming for MPEG devices
F	Key file system, used for storing private keys
I	Internet protocol device, used to implement IP communications
l	Ethernet device, used to implement Ethernet communications
M	Mount driver, used to import remote file systems
p	Prog device, provides query and control of status of Inferno system processes
r	Real-time clock device, controls real-time clock and NVRAM access
s	Limbo file channel server

Table 5-1 Inferno System Devices—*Continued*

t	Serial communications device, used to implement serial communications
U	Host OS file system
V	Television, provides control for ITV applications

The kernel maintains a table that maps these device characters to device-specific operations for each type device supported by the kernel. A kernel device driver is a server in the sense of the Styx protocol, with the messages implemented as local rather than remote procedure calls.

There are 14 device switches that implement the calls contained in the system module, *sys.m*. Each device switch has a direct counterpart in the Styx messages that implement remote procedure calls. A synopsis is presented in Table 5-2.

Table 5-2 Device Switches

init	Initializes the device driver
attach	Creates a channel attached to the root directory of the device
walk	Traverses a node in the directory tree supplied by the device
clone	Duplicates a channel
close	Closes a channel
create	Creates a file
open	Prepares a file for I/O
remove	Removes a file

Table 5-2 Device Switches—*Continued*

read	Returns data from a file
bread	Performs a block read on a file
write	Writes data to a file
bwrite	Performs a block write to a file
stat	Returns the attributes of a file
wstat	Writes the file attributes

Device drivers for specific hardware, such as an MPEG card or television interface, must be implemented when the Inferno kernel is ported to a new hardware environment. Inferno system installations in an emulation environment require file system interfaces and interface programs to connect to local device drivers and the file system implemented by the host operating system.

Namespace Construction

An Inferno namespace or file system can consist of files that represent devices or processes on remote systems.

The file system model is used for the Inferno namespace because files, file structures, permissions and access controls are well understood. The files interfaces of *open*, *read*, *write*, *seek*, and *close* **(**implicit**)** are well known. Naming conventions, protection, and multiplexing are provided automatically by a file interface. The file interface allows the Inferno system to provide a single remote procedure call like protocol to access resources. The file model allows resources to be operating system independent and it allows multiple programs to access a server process.

There are several ways that a namespace can be constructed in the Inferno system. This namespace can be created by the application, by another

program, or outside the application by an administrator. As far as the application is concerned, the files are local. The application does not have to be concerned about where the resources are located. A program can assemble its own namespace, load its namespace from a file, or have its namespace created by user-level commands.

For each related group of processes, the kernel maintains a table that translates a user file descriptor into a channel structure, or *Chan*, on which file-based system calls operate. The channel structure is similar to the Unix kernel file structure and is central to building namespaces in the Inferno system. A file descriptor, passed as a parameter in a system call, is converted to a channel structure to provide the kernel with a representation of the file object. The kernel stores a list of bindings between channels in an internal mount table.

The namespace provides a customized, local view of resources available in the network to an application. The correct functioning of the Inferno system depends on the construction of a useful namespace. Building a namespace is a two step process using the namespace operations *mount* and *bind*.

Network transparency is supported for all Inferno system files including files of data, devices, and daemon interfaces. Applications are networked without explicit network programming.

Inferno system networking is more than simple client/server computing. The roles of “client” and “server” are relative to some set of shared resources. In general, application problems can be solved by having two or more machines cross mount resources and have processes on each coordinate on using the combined resources.

Mount

Mounting a namespace component provides local access to it. Local devices are preceded by the pound sign and are typically pre-mounted. Typically, they are mounted during system initialization, although they can be mounted through user-defined scripts. Remote namespace components, such as remote file systems, are explicitly mounted by an application using the *mount* call from *system.m*. For example, the Limbo code fragment:

```
if(sys->mount(fd, "/n/client", sys->MREPL, "") < 0) {
        sys->fprint(stderr, "exec: mount: %r\n");
        return;
        }
```

attaches *fd* to the */n/client* directory in the current namespace. The *MREPL* flag hides the current */n/client* by replacing it in the local file tree. Other flags provide for the construction of union directories that behave like the concatenation of the component directories with a specified order. The null string final parameter selects the default tree on the server.

The mount operation adds a file tree from a server machine to the current namespace (see Figure 5-3). Once the file tree is attached, its members can be accessed in the same manner as local files (see Figure 5-4). Applications cannot detect that those files are remote. This is true even for *fileC*, a file that this server has mapped into its namespace from yet another server.

Figure 5-3 Client/Server View Before Mount

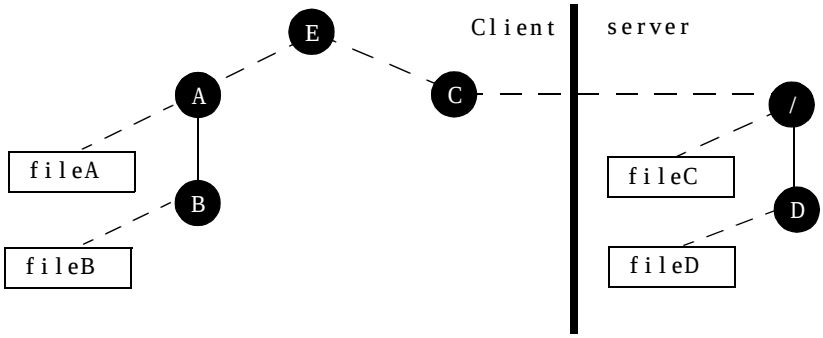
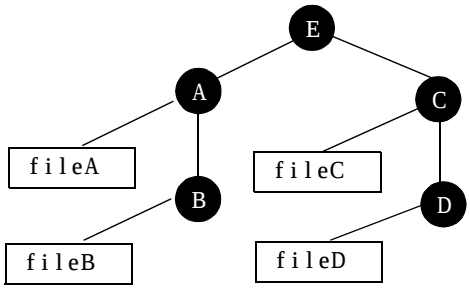


Figure 5-4 View After Mount, Before Bind



bind

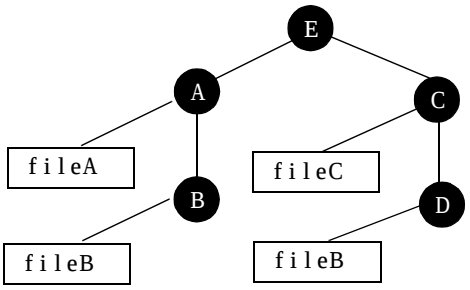
Binding duplicates some piece of an existing namespace to another point in the namespace. That is, the *bind* call places a mounted namespace component in a desired location.

The bind operation maps one file name in the namespace to another. For example, an application may expect to find a certain file, *fileB*, in directory, *D*, but the file actually resides in directory B. See Figure 5-4.

The bind operation can be used to make fileB appear to reside in directory *D*. See Figure 5-5.

Note that *fileB* is still available from directory *B*. In this case, the original contents of directory *D* are obscured. (Note that the contents of directory *D* are not replaced and they will be available again after an *unmount*. Different bind options can be used to make *D* a union directory that has both the original contents and those from directory *B*. An example using that will be shown later.

Figure 5-5 File View After Bind Operation



Managing Complexity

One implication of this example is that applications become simpler because they need not cope with a variety of configurations. They can be “hard-coded” to look for resources in certain directories. Ordinarily, this would be bad practice; however, in the Inferno system, there can be a

separate set-up stage to provide the application with the needed file (possibly using mounts over the network) in the expected location.

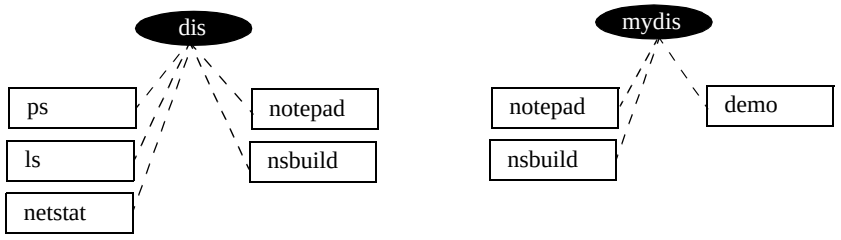
Separating configuration issues from application concerns makes each task easier. This leads to better software that is less expensive to develop and maintain.

Namespace Customization

An important facility of Inferno namespace operations is the ability to create “union” directories. That is, the ability to make several directories appear as one. Member files of each contributing directory appear in the union. When contributing directories have file names in common, the file available from the union depends on the sequence of constructing the union.

In the Inferno system, the management of commands available to the Inferno shell is done through the use of “union directories”. The Inferno shell has no PATH variable. That is, it does not maintain a list of directories to be searched for commands. The Inferno shell is programmed to look for executable files in the following order: 1) the explicitly named path, when given, 2) the current directory, and then 3) the <*inferno_root*>*/dis* directory. See Figure 5-6.

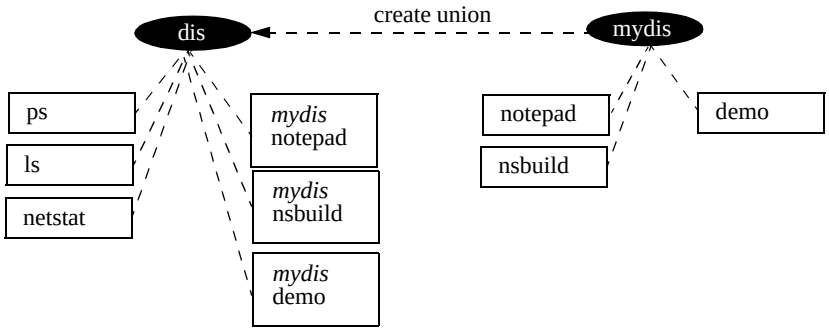
Figure 5-6 Namespace Before Customization



When a user wants access to a command that does not exist in */dis*, the user can bind the command into */dis*. If a user wants to use a customized version of a command already existing in */dis*, the user can bind in the new command so that the personal version takes precedence over the standard version. See Figure 5-7.

The Inferno system provides an operation to clone an existing namespace to obtain an independent copy. Changes in the copy do not influence those using the original.

Figure 5-7 Namespace After Customization



Controlling Access to Resources

System security can be enhanced by namespace operations that shrink an existing namespace. A process can be restricted so that it can only access a limited set of files and, once restricted, the process can access no others.

This technique is applied before starting a test program that requires a carefully controlled environment or before starting a suspicious program that might harm the local file system. The technique, also, can be applied on Inferno server systems to restrict the set of files available for mounting by clients.

Controlling access to resources by namespace restrictions is an adjunct to the control provided by file and directory permissions. For example, a mail reader application program might need a user's identity to access some of that user's files. Should satisfying that requirement imply that the program have the potential of accessing any of that user's files? By appropriately restricting the namespace, that need not be the case. The program can be restricted to a universe containing only the files it needs. The program's permission to access other files is irrelevant if pathnames to those other files cannot be defined.

Recall that files are representing all sorts of resources. By restricting the namespace, the application's universe of resources is limited. Omitting the network interface means that it cannot make network calls. Omitting the interface to running programs means that the application cannot terminate other applications.

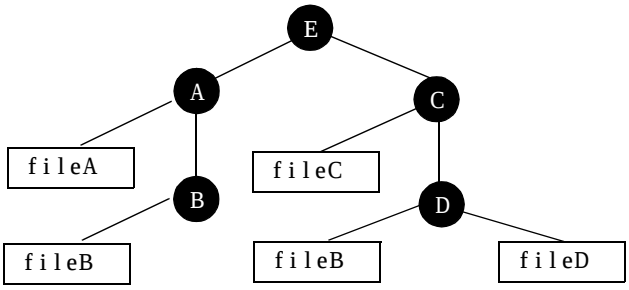
Union

When *bind* is used with the *-a* or *-b* options, the effect is not to replace the target but to create a union of the contents of the source and target at the target directory. The instruction

```
bind -b A/B C/D
```

results in a union of the contents of two directories, as shown in Figure 5-8.

Figure 5-8 Union of the Contents of Directories A/B and C/D



The *bind* operation can be repeated to create the union of several directories at *D*.

unmount

The effects of binding and mounting are undone by the *unmount* call.

Limbo Programming Language

Limbo is a general-purpose programming language used to create Inferno applications in a simple and efficient way. It can be used to produce applications for the Inferno environment as well as to create gateways and interfaces to services not native to the Inferno system. The language is designed to take advantage of the network and security features of the Inferno system.

Limbo is a modular, procedural language, with a familiar C-like syntax, that is compiled into a hardware independent representation, or byte code, for execution on a virtual machine. It is strongly typed, provides automatic garbage collection, and supports only very restricted pointers.

Limbo applications can include just-in-time compilation at the discretion of the programmer. Alternatively, a Limbo application can be interpreted in the Dis virtual machine. Limbo applications run on any platform that supports the Inferno API.

Limbo is discussed in more detail in the *Inferno Programmer's Guide*.

Modules

A Limbo program is a module, or set of modules, that performs a task. In source form, a module consists of a module declaration that specifies the public interface—the functions, abstract data types, and constants that the module makes visible to other modules—and an implementation that provides the actual code. By convention, the module declaration is placed in a separate *.m* file so it can be included by other modules, and the implementation is stored in a *.b* file. Modules can have multiple implementations, each in a separate implementation file.

At run time, modules are loaded dynamically, enabling the Inferno system to keep programs small. The module load statement fetches the code and

performs run-time type checking. Once a module has been loaded, its functions can be called.

Limbo is strongly typed, and programs are checked at compile time and again when modules are loaded. The Limbo compiler compiles each source file into a machine-independent byte-coded *.dis* file that can be loaded at run time.

Functions and Variables

Functions are associated with specific modules, either directly or as members of abstract data types within a module. Functions are visible outside their module only if they are part of the module interface. If the target module is loaded, specific names can be used in a qualified form such as *sys->print* or without the qualifier if imported with an explicit *import* statement.

Besides normal block structure within functions, variables can have global scope within a module. Module data can be accessed using the module pointer.

Data

The size and signedness of integral types are specified in Limbo, and are the same everywhere. Character constants are enclosed in single quotes and can use escapes like *\n* or *\udddd*, but the characters themselves are in Unicode and have type *int*. There is no enumeration type, but there is a *con* declaration that creates a named constant.

Limbo also provides Unicode strings, arrays of arbitrary types, lists of arbitrary types, tuples (unnamed structures with unnamed members of arbitrary types), abstract data types or *adt*'s (named structures with function members as well as data members), reference types (restricted

pointers that can point only to *adt* objects), and typed channels (for passing objects between processes).

A channel is a mechanism for synchronized communication. It provides a place for one process to send or receive an object of a specific type. The attempt to send or receive blocks until a matching receive or send is attempted by another process. The *alt* statement selects randomly but fairly among channels that are ready to read or write. The *spawn* statement creates a new process that, except for its stack, shares memory with other processes. Processes are preemptively scheduled by the Inferno system. (The Inferno system processes have much in common with threads in other operating systems.)

Limbo performs automatic garbage collection, so there is no need to free dynamically created objects. Objects are deleted and their resources freed when the last reference to them goes away. In general, the release of resources happens immediately (“instant free”), whereas the release of cyclic data structures can be delayed.

Operators and Expressions

Limbo provides many of C’s operators, but there is no *?:* operator. Pointers, created with *ref*, are very restricted and there is no *&* (address of) operator. There is no address arithmetic and pointers can only point to *adt* objects. Array slicing is supported, however, and conveniently replaces many pointer constructions.

There are no implicit coercions between types, and only a handful of explicit casts. The numeric types *byte*, *int*, and so on, can be used to convert a numeric expression, as in

```
nl := byte 10;
```

and *string* can be used as a unary operator to convert any numeric expression or array of bytes to a string (in *%g* format).

Statements

Statements and control flow in Limbo are similar to those in C. A statement is an expression followed by a semicolon, or a sequence of statements enclosed in braces. The similar control flow statements are

```
if (expresssion) statement
if (expression) statement else statement
while (expresssion) statement
for (expresssion; expresssion; expresssion) statement
do stat while (expresssion);
return expresssion;
exit;
```

The *exit* statement terminates a process and frees its resources. There is also a case statement analogous to C's *switch*. It also supports string and range tests. A break or continue followed by a label causes a break out of, or the next iteration of, the enclosing construct that is labeled with the same label.

Comments begin with # and extend to the end of the line. There is no preprocessor, but an include statement can be used to include source code, usually module declaration files.

Libraries

Limbo has a growing set of standard libraries, each implemented as a module. A handful of these (notably *Sys*, *Draw*, *Math*, and *Tk*) are included in the Inferno kernel because they will be needed to support almost any Limbo program. Among the others are *Bufio*, a buffered I/O package, and *Regex*, for regular expressions.

Dis Virtual Machine

The Dis virtual machine provides the execution environment for programs running under the Inferno operating system including Limbo applications. The virtual machine models a CISC-like, three operand, memory-to-memory architecture that makes the Inferno system flexible and portable. Code can either be interpreted by a C library or compiled on-the-fly into machine code for the target architecture.

For simplicity, this section uses the term *thread* to describe a thread of control created by the Limbo *spawn* statement. The terms *process* and *OS process* are used for Inferno processes and, in a hosted environment, host operating system processes. There are separate schedulers for operating system processes and Limbo threads.

Dis can schedule multiple threads that are executed in round-robin fashion. This enables multi-tasking even on thin appliances.

Memory Organization

Memory for a thread of execution is divided into several separate regions:

- The *code segment* stores either a decoded virtual machine instruction stream suitable for execution by the interpreter or native machine code that is compiled on-the-fly for the host CPU. Neither type of code segment is addressable from the Dis instruction set.
- A thread executing a module has access to two regions of data memory:
  - A module pointer (*mp*) defines a region of global storage for a particular module

- A frame pointer (*fp*) defines the current frame activation record for the thread

  Frames are allocated dynamically from a stack by internal *call* and *return* instructions. The stack is extended automatically from the heap. The *mp* and *fp* registers cannot be addressed directly and can be modified only by *call* and *return* instructions.

- Memory can be allocated from the heap using various forms of *new* instructions and can subsequently be addressed using a double indirect addressing mode with the pointer stored in either *fp* or *mp*.

Data memory is addressed as bytes. Words are stored in the native representation of the host CPU. Data types larger than a byte must be stored at aligned addresses.

Only a single instance of Dis controls the heap. Multiple instances cannot share memory, since there are no locks on the heap.

All modules and threads share a single data heap. Program module instances and stack frames are allocated from the heap. Each heap object is associated with a type descriptor that contains its size and the location of pointers to other heap objects. Memory location information is transmitted in the form of type descriptors that are generated automatically by the Limbo compiler.

When a new object is allocated, all of its pointers are initialized to *nil*, a value guaranteed to cause an exception if dereferenced. Memory protection is enforced by cooperation between the compiler and the virtual machine.

Garbage Collection

The garbage collector in the Inferno Dis virtual machine is a hybrid scheme that uses two algorithms:

- Reference counting
- Real time mark-and-sweep for cyclic structures

Most memory is collected immediately using a reference counting algorithm. Dis performs reference counted garbage collection to restore resources to availability immediately after last use. Every memory location that stores a pointer is known to the virtual machine and memory can be initialized and deallocated quickly and correctly.

Data structures that are *cyclic* are detected by the garbage collector that runs as an independent background process in the virtual machine. Cyclic structures are collected by a real time mark-and-sweep process using a new four-color algorithm to ensure efficient deallocation. Since the mark-and-sweep collection only involves cyclic structures, it is rarely invoked and incurs virtually no overhead.

The hybrid approach for garbage collection allows code to be generated in several styles: pure reference counted, mark and sweep, or a hybrid of the two approaches. Compiler writers have the freedom to choose how specific types are handled by the machine to optimize code for performance or language implementation. Instruction selection determines which algorithm will be applied to specific types.

When using reference counting, pointers are a special operand type and should only be manipulated using the pointer instructions in order to ensure the correct functioning of the garbage collector. Every memory location that stores a pointer must be known to the interpreter so that it can be initialized and deallocated correctly. The information is transmitted in the form of type descriptors in the object module. Each

type descriptor contains a bit vector for a particular type where each bit corresponds to a word in memory. Type descriptors are generated automatically by the Limbo compiler. The assembler syntax for a type descriptor is:

```
desc   $10, 132, "001F"
```

The first parameter is the descriptor number, the second is the size in bytes, and the third a pointer map. The map contains a list of hex bytes where each byte maps eight 32 bit words. The most significant bit represents the lowest memory address.

A one bit indicates a pointer in memory. The map need not have an entry for every byte and unspecified bytes are assumed zero.

Operand Size

Operand sizes are defined as follows: a byte is 8 bits, a word or pointer is 32 bits, a float is 64 bits, a big integer is 64 bits. The operand size of each instruction is encoded explicitly by the operand code. The operand size and type are specified by the last character of the instruction mnemonic:

Table 5-3 Operand size

W	word, 32-bit two's complement
B	byte, 8-bit unsigned
F	float, IEEE format
L	big, 64-bit two's complement
P	pointer
C	Unicode string encoded in UTF-8
M	memory
MP	memory containing pointers

Effective Addresses

Each instruction can potentially address three operands. The source and destination operands are general, but the middle operand can use any address mode except double indirect. If the middle operand of a three address instruction is omitted, it is assumed to be the same as the destination operand.

The general operands generate an effective address from three basic modes: immediate, indirect and double indirect. The assembler syntax for each mode is:

10(fp)	30-bit signed indirect from fp
20(mp)	30-bit signed indirect from mp
$0x123	30-bit signed immediate value
10(20(fp))	two 16-bit unsigned offsets double indirect from fp
10(20(mp))	two 16-bit unsigned offsets double indirect from mp

Program Execution

Multiple Limbo threads are placed onto a run queue and executed in round-robin fashion. Threads are multiplexed onto processes; a given thread can be executed by several processes in sequence before it completes, and multiple threads can be attached to a single process.

Table 5-4 lists the states that threads are placed in by Dis.

Table 5-4 Thread States

alt	Inter-thread communication processing
broken	Thread has crashed
delete	Remove from queue
exiting	Thread instructions completed
ready	Available to execute instructions
receive	Ready to receive a value from another thread
release	Remove from queue to complete a kernel call
send	Ready to transmit a value to another thread

Execution continues for a given thread until one of the following conditions is met.

- The thread is completed and assigned an *exiting* state by Dis (Figure 5-9, Step *A*).

  A thread will also be released from the virtual machine if it is broken.

- The thread completes its quantum.

  When a thread executes a specified number of instructions, it is removed from virtual machine execution and placed on the Limbo thread ready queue (Figure 5-9, Step *B*). The ready queue is a linked list of threads that are waiting for access to the CPU.

- The thread must send a message to, or receive a message from, another thread.

  A *send* state (Figure 5-9, Step *C*) requires a *receive* message from another thread (Figure 5-9, Step *D*) to place it back into the ready queue. When a thread is placed into a *send* state, it will not be placed back on to the queue until the thread that is to receive its message accepts it. Similarly, a thread in a *receive* state is not placed back on to the queue until it has received the message and acknowledged it to the sender.

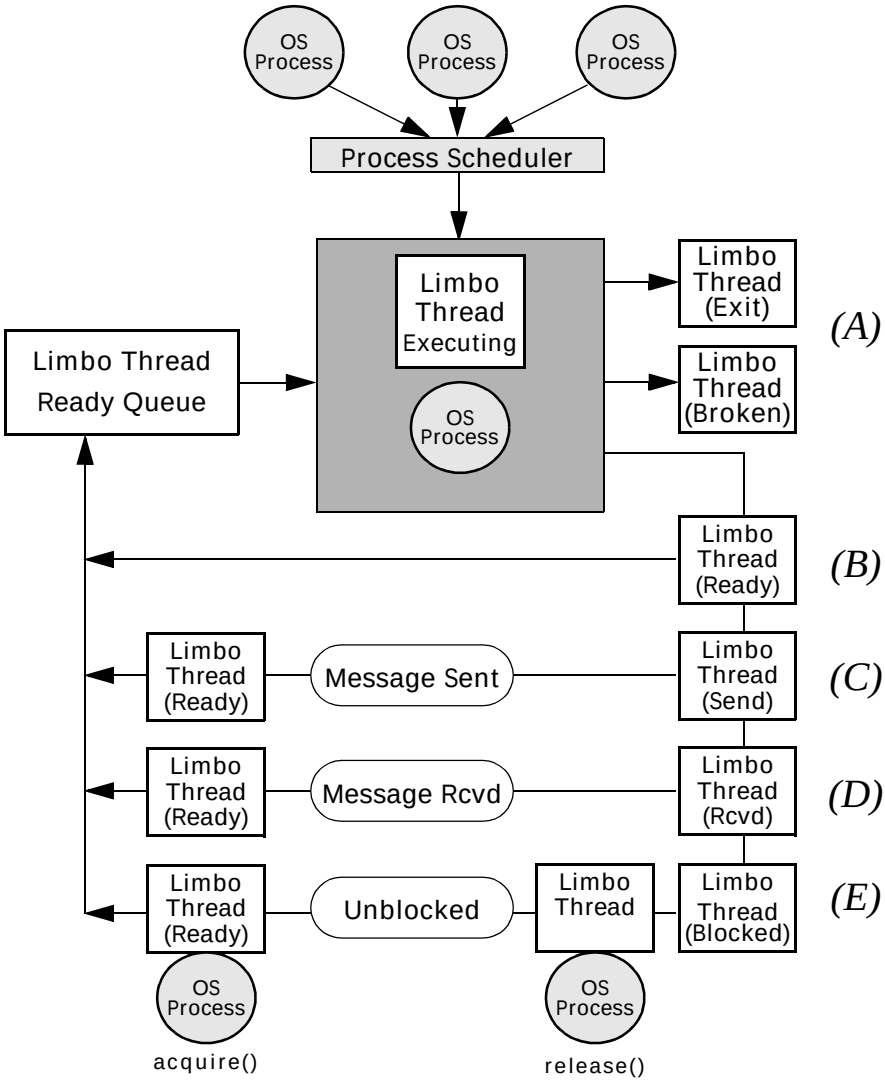
  The *alt* state exists when multiple messages must be exchanged among threads. The thread is removed from the ready queue and the *alt* routine coordinates multiple *send* and *receive* states across multiple channels.

- The thread is blocked (Figure 5-9, Step *E*).

  A thread is blocked when a file manipulation operation or a network access instruction is encountered. A *sleep* instruction in a host operating system also blocks a thread. Blocked threads are removed from execution when a *release* request is issued. The operation that caused the block condition is then scheduled by the operating system.

  On completion of the blocking operation, the operating system process issues an *acquire* call to signal Dis that the Limbo thread is ready to continue execution. When acquired, the thread is placed at the front of the ready queue.

Figure 5-9 Dis Virtual Machine Operation



Styx Communications Protocol

This section describes the Inferno networking environment, including an overview of the Styx communications protocol, which is used for passing messages between clients and servers across the network. Most of the network traffic between Inferno systems is in the form of Styx messages. Most applications do not use the Styx protocol directly, but make calls to library routines that access files. Applications do not need to explicitly manage details of the Styx protocol. File operations that invoke Styx result in messages that perform the necessary operations on the remote namespace. The complexity is concealed from the developer, who simply invokes *mount*, *open*, *read*, and similar system calls as necessary.

An application can use something other than Styx for communication if it does not need to make use of a namespace. For example, Charon, the Inferno web browser, and Ftp Access are Inferno applications that are part of the Inferno Window Manager, *wm***.** They use the Inferno TCP/IP device drivers to communicate with other systems.

Because one of the goals of the Inferno system is to encourage resource sharing in a widely distributed environment, the organization of the networking environment plays a crucial role in the success of the system. Inferno system users can access the system using a wide variety of hardware: PDAs, set-top boxes, terminals, personal computers, or workstations. To facilitate resource sharing in a diverse environment, each type of hardware device must present the same view of the network to the user.

Like other services in the Inferno system, network transports, network devices and network connections are represented as file systems.

An Inferno *server* is a machine that provides access to one or more hierarchical file systems on request from a client, usually through a network connection. A *connection* to a server is a bidirectional

communications link between a server and a client. A connection remains established while any of the files in the connection directory are referenced or until a *close* system call is received from the network. The primary function of the server is to process file system navigation and file manipulation requests from its clients.

Using the same set of files to represent different devices allows the creation of common tools that can serve several networks or interfaces. Some familiar services whose implementation on other operating systems is more complex can be provided just by echoing the contents of the interface files. Network connections represented by these files behave the same way for all networks and allow application programs to contain no network-specific code.

The Inferno kernel, either hosted or native, supports the Internet Protocol (IP) suite, including Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

Styx Protocol

File operations between any two machines are accomplished by sending requests and receiving replies using the Styx protocol in a manner similar to Unix remote procedure calls. The protocol supports remote access to files and allows a remote machine to use these interfaces as gateways.

Styx messages are not manipulated directly by an application designer. Any file operation that invokes Styx results in messages that perform the necessary operations on the remote namespace. The exact set of messages differs depending on the call, but in all cases the complexity is concealed from the developer, who simply invokes *mount*, *open*, *read* and similar system commands as necessary.

The protocol contains messages that perform:

- Navigation over the file hierarchy
- Control of file attributes
- Access to data in files
- Miscellaneous functions

Operation

A kernel-resident file server, *devmnt*, functions as a mount driver that is used by the *mount* system call to connect the namespace of a process to the service provided by a server over a communications channel. After the *mount* has executed successfully, system calls involving files in that portion of the namespace will be converted by *devmnt* into the appropriate Styx messages to the server.

The *mount* system call issues *attach* messages to the server to identify and validate the user of the connection. Each distinct use of a connection must mount it separately; *devmnt* multiplexes the access of the various users and their processes to the service.

File oriented system calls are converted by the kernel into Styx procedure calls. The server *devmnt* translates these procedure calls into remote procedure calls, which are then transmitted to the server as Styx messages over the communications channel. Each message is implemented by writing the protocol message to the server followed by reading the server channel to get the reply. Errors in the reply message are turned into system call error returns.

A *read* or *write* system call on a file served by *devmnt* can be translated into more than one message since there is a maximum data size for a Styx message. The system call will return when the specified number of bytes have been transferred or a short reply is returned.

Styx Messages

A descriptive listing of Styx message pairs by general category is presented in the following tables.

Table 5-5 Styx Message Pairs

Tattach Rattach	Authenticate the caller, establish fid that points to the root directory of the namespace of the machine that was called
Tclone Rclone	Create a copy of an existing fid; when done, two fids are pointing to the same file in the hierarchy
Twalk Rwalk	Move fid one step in the file hierarchy
Tclunk Rclunk	Close a fid (throw it away); the actual file is unaffected
Tstat Rstat	Return file attributes
Twstat Rwstat	Set attributes; usually, a Tstat is first done to get the attributes, and then a Twstat is executed with a status buffer altered as needed
Topen Ropen	Check file permissions and, if allowable, prepare an existing file for reading or writing
Tcreate Rcreate	Create and open a new file
Tread Rread	Access contents of file
Twrite Rwrite	Write to file

Table 5-5 Styx Message Pairs

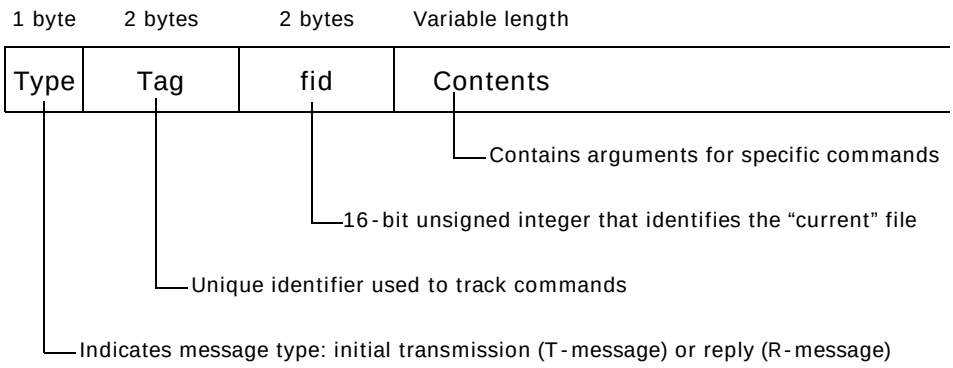
Tremove Rremove	Remove the file that the fid is pointing to and clunk the fid
Tnop Rnop	No operation; can be used to synchronize the channel between two services
Tflush Rflush	Interrupt a pending operation
Rerror	Indicates that an error has occurred; the Contents segment contains the text of the error generated by the server

Styx Message Structure

Styx runs on top of any reliable transport level protocol so that the underlying network connection is transparent to applications. Styx relies on several properties of the underlying transport protocol. For example, it assumes that messages arrive reliably and in sequence.

The general structure of Styx messages is given in Figure 5-10. Each message consists of a sequence of bytes in defined byte order for machine independence. The first byte, *Type*, is the message type that is one of the enumerated constants defined in *styx.h*. Message types are typically arranged in pairs, called transactions. T-messages are requests from client to server (for example, *Topen*) that are matched by corresponding R-messages that are replies from server to client (for example, *Ropen*).

Figure 5-10 General Structure of a Styx Message



The second field, *Tag*, is a chosen by the client to identify the message. The client must ensure that no two outstanding messages on the same connection have the same tag. The reply type will have a value one greater than the request that generated it except in the event of an error reply (*Rerror*).

The *fid* field is a file identifier that is part of Styx messages that operate on files. Every file that is manipulated by the operating system is identified by a *fid*. The *Contents* field contains one or more elements that represent the information required by the message.

Inferno Networking

Any Inferno system configuration can be made up of many distributed clients and servers. the Inferno system does not dictate any rules or restrictions on where these machines are located – they can be in the next room or across the country. Additionally, Inferno-based machines can be connected to each other by a variety of physical media and network protocols as required to maximize connectivity and throughput.

The Inferno system hides network complexity from applications. When an application needs to access services that reside on remote servers, a few simple operations are used to construct a single, locally-represented namespace that can encompass multiple machines and multiple networks. For example, if a client has access only to a TCP/IP network and requires access to other networks, the client can mount the additional networks from a server that has access to them. When attached to the local namespace, the remote server networks appear to be local to the client.

Like other services in the Inferno system, network interfaces are represented as file systems in a local namespace. The Styx communications protocol unifies different parts of the local namespace that can be constructed from the vast collection of files on the network. File operations between any two machines are accomplished by sending requests and receiving replies using Styx messages. The protocol supports remote access to files and allows a remote machine to use these interfaces as gateways.

Styx messages are not manipulated directly by an application developer. Any file operation generates the appropriate Styx messages to perform the necessary operations on the namespace. The exact sequence of messages differs depending on the call, but the complexity is concealed from the developer who simply invokes familiar *open*, *read*, and similar file system operations.

Styx runs on top of any reliable transport level protocol, making the underlying network connection transparent to applications. Styx relies on several properties of the underlying transport protocol. For example, it assumes that messages arrive reliably and in sequence.

Inferno-enabled Systems

On Inferno-enabled systems in the network, Limbo programs can communicate with each other through Styx. You can use *mount* commands to access files and devices on those systems. Non-inferno enabled systems cannot communicate in this way.

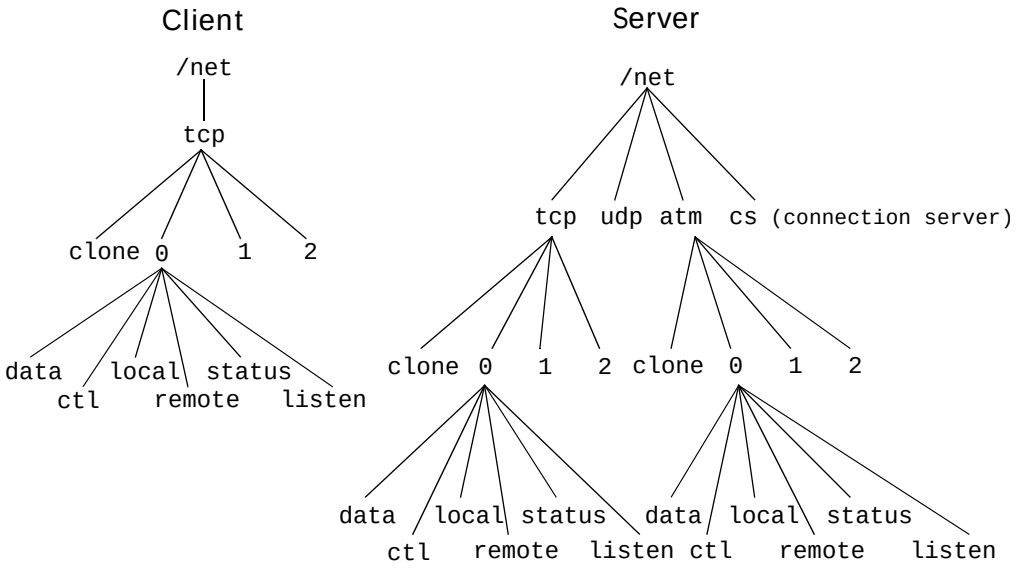
However, communication between Inferno-enabled and non-Inferno systems is possible in a limited way using the HyperText Transfer Protocol (HTTP) and the File Transfer Protocol (FTP). FTP Access is an application written in Limbo that is delivered with the Inferno system. It uses FTP to access files on a remote system that is not necessarily Inferno-enabled. You are limited to the functionality provided by these protocols; you are not able to use all the Inferno system functionality.

Establishing a Network Namespace

A server file system is incorporated into a client namespace using the kernel *mount* and *bind* calls. For example, if a client has access only to a TCP/IP network and requires access to other networks, the client can *mount* these networks from a server that has access to them by constructing a union of the client and server directories by default named */net*.

Figure 5-11 shows the */net* file tree for a client that only has access to a TCP/IP connection. The */net* file tree on the server called <*server*> has access to multiple connections.

Figure 5-11 An Example of Two Network File Trees



The client creates a namespace that incorporates the */net* file system from the server by issuing a *mount* and *bind* sequence of commands similar to the following from the Inferno control console:

```
mount net!server /n/remote
bind -a /n/remote/net /net
```

After the *mount* and *bind* commands have executed successfully, all of the listed networks that are connected to the <*server*> are available to the client. The client can then send requests intended for these networks by way of the server. The server acts as a gateway without the need for a client protocol stack.

Following the *mount* and *bind* sequence described above, a list of directories at the client machine would look like the following:

```
$ ls /net
cs
atm
udp
tcp
tcp
```

Note that there are two */net/tcp* directories resulting from the union of the local and remote directory structures. Both the local and remote versions of this directory are part of the local namespace. Only the first, on the local machine, is accessible. Using the *-b* option in the *bind* command would make only the remote */net/tcp* directory accessible.

Network Connection Server Applications

A client with access to a server may be able to use all of the networks attached to that server.

The Inferno server application named *cs* (connection server) hides the details of call setup from applications. *Cs* is provided as a demonstration server application.

A network connection is established in two phases. The *cs* application is first contacted to translate a logical network name into a device name and a connect address string. The *cs* service receives a logical name such as *tcp!<server>!styx* and returns the device name and connect address string:

```
/net/tcp/clone
connect 35.104.9.52!6666
```

The path name identifies the network device, and the connect address string is used to set up a connection. To establish the connection, the client then opens */net/tcp/clone* and writes the control message into it.

When */net/tcp/clone* is opened, it actually returns a file descriptor to

/net/tcp/<*number*>/ctl

where <*number*> is a numeric directory name (see Figure 5-11). Text commands are then written to the *ctl* file to control the connection.

Each network directory contains a *clone* file and a directory for each connection of that type, numbered 0 through *n*, where *n+1* is the number of connections of a single type. Opening the *clone* file finds and reserves an unused connection directory and opens its *ctl* file. The file descriptor returned to the client by the *open* call points to the *ctl* file. Each connection directory contains files to control a single connection and to send and receive information to it.

TCP Connections

TCP is the most common protocol used in the Inferno system. A TCP connection is represented by the files indicated in Figure 5-11 under the */net/tcp/0* directory. The organization of the TCP file system is similar to that of other network connections types. The function of each of these files is described in the following sections.

Control File

Reading the *ctl* file descriptor returns a text string containing the connection number. The client program uses this value to construct the full name of the connection directory. The client program can *read* and *write* the file descriptor returned by the *open* call to send and receive control messages to the driver or connection. A connection remains

established while any of the files in the connection directory are referenced, or until a *close* message is received from the network.

Data File

The *data* file provides access to the media. Reading the data file returns data received across the network, blocking the file until some data is available. Writing to the *data* file prefixes a header and queues data for transmission.

Status File

The *status* file can be read to yield protocol-dependent information about the state of the interface.

Local and Remote Files

The files *local* and *remote* contain an IP address and port number for each end of the connection.

Listen File

The *listen* file is used to accept the incoming calls from the network. If a process opens the *listen* file, the process will block until an incoming call is received.