2007

# DeepFreeze: a management interface for ISEAGE

Nathan Lyle Karstens
*Iowa State University*

**DeepFreeze: a management interface for ISEAGE**

by

**Nathan Lyle Karstens**

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Co-majors: Information Assurance; Computer Engineering

Program of Study Committee:
Doug W Jacobson, Major Professor
Thomas Earl Daniels
Johnny S Wong

Iowa State University

Ames, Iowa

2007

# TABLE OF CONTENTS

# LIST OF FIGURES

# ABSTRACT

DeepFreeze is a framework designed to provide a unified command and control interface for the ISEAGE project. Besides being a graphical interface, it includes an extensive API and methodology for authoring and controlling applications executing within ISEAGE. It utilizes a wide array of technologies in the computing field to provide functions such as power management, efficient one-to-many communication, and fault tolerance for executing processes.

A concept application designed to aid in file distribution is presented. Also discussed are the modifications that were necessary to migrate the Traffic Mapper, a foundational component of ISEAGE, to the DeepFreeze environment.

# CHAPTER 1. INTRODUCTION

DeepFreeze arose from a need to provide a unified command and control interface for the ISEAGE project. On one level, it seems to be simply a graphical interface. However, this aspect of DeepFreeze operates on an extensive API and methodology for authoring and controlling applications executing within the ISEAGE environment. It is intended both to assist the user in monitoring and controlling ISEAGE applications, and to aid the developer in creating those applications.

DeepFreeze has four goals: 1) provide a unified management interface for control of ISEAGE computers, 2) provide a degree of fault tolerance by monitoring applications running within the ISEAGE environment, 3) facilitate communication between applications and their controlling interface, and 4) aid developers in authoring ISEAGE applications.

Although individual projects within ISEAGE have been completed, they have not been integrated into a central command console. It is hoped that providing a framework for this integration will encourage developers to author control components for their project, thereby changing the ISEAGE project's culture from a group of individuals working on separate projects to a team of individuals contributing to the vision behind ISEAGE.

This chapter will briefly discuss the design of ISEAGE and introduce DeepFreeze. Attention will be given to several technologies utilized by DeepFreeze that are perhaps not universally understood within the computing field.

## 1.1 ISEAGE Architecture

A basic understanding of ISEAGE's architecture and intended functionality is requisite to understanding the architecture and design decisions behind DeepFreeze. The

ISEAGE network emulation and computing cluster is composed of 64 computers, divided into four racks—"Icles"—of 16 (see Figure 1). As these computers do not have attached hard drives, each Icle contains an additional computer that provides a file system through an NFS mount. This occurs on a dedicated management network connected to device em0 on each board. The file server also acts as a router; it connects its Icle to the management console and the Internet.

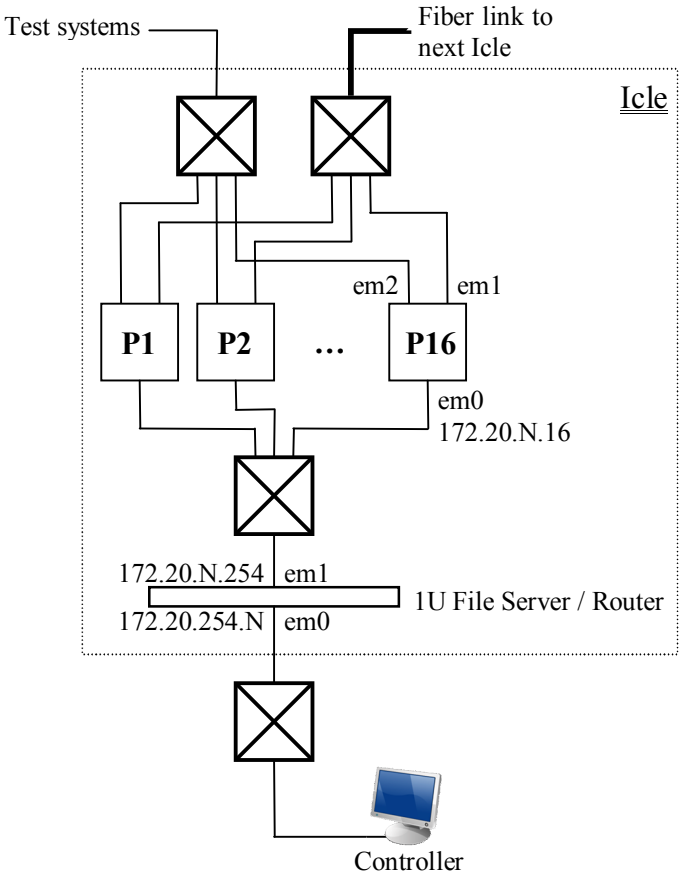


Figure 1: ISEAGE Architecture

Each computer contains an additional pair of network interfaces (em1 and em2) used by the Traffic Mapper to provide a number of features related to packet routing and Internet-scale network emulation. These interfaces are controlled exclusively by the Traffic Mapper and are not assigned IP addresses.

Both the ISEAGE computers and their file servers run FreeBSD version 5.3. This operating system provides, among other things, a robust networking stack utilized heavily by DeepFreeze.

## 1.2 DeepFreeze Architecture

Fulfilling the goals behind DeepFreeze required the design of a fairly unique architecture (see Figure 2). The overall system is the combination of four types of software: 1) the DeepFreeze Console, 2) DeepFreeze plugins, 3) the DeepFreeze Daemon, and 4) remote applications. A remote application and its plugin are collectively referred to as an application.

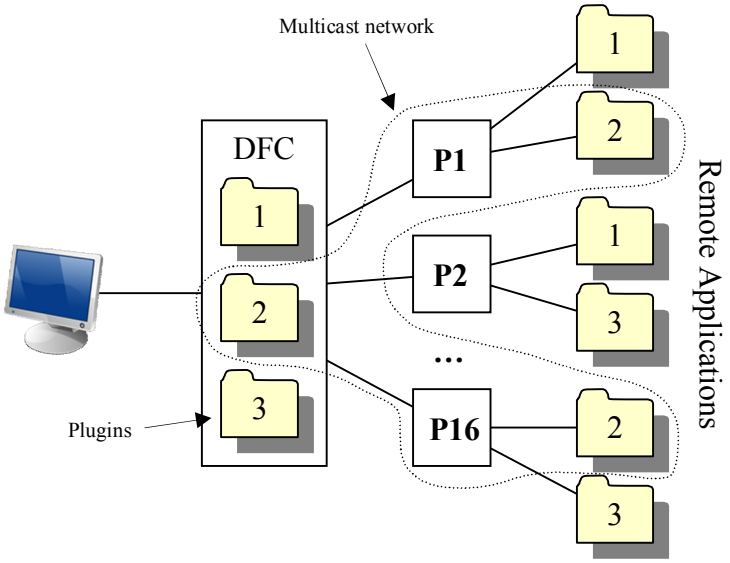


Figure 2: DeepFreeze Architecture

The DeepFreeze Console (DFC) is a GUI application that runs on one or more client computers. In addition to providing an interface for several management functions, it also displays DeepFreeze plugins, which are the control interfaces provided for each application running within the ISEAGE environment. Plugins are compiled as shared libraries and are dynamically loaded by the DFC.

The DeepFreeze Daemon (DFD) is a system daemon (background program) responsible for loading and communicating with remote applications, such as the Traffic Mapper or DeepFreeze's Distributed File System (DFS).

Communication between plugins and remote applications is facilitated through the insertion of software between the transport and application layers in the TCP/IP network model. In the DFC, this software takes the form of an API passed into a plugin upon initialization. The DFD software utilizes local sockets and file descriptor manipulation. Monitoring functionality is provided by the DFD through the insertion of a conceptual layer between the operating system and remote applications.

In order to provide additional management channels, DeepFreeze offers two types of communications to both remote applications and their plugins: 1) in-band and 2) out-of-band. In-band communications are any messages intended to be passed between a remote application and its plugin. Out-of-band communications are messages sent directly to the user from either a remote application or its plugin. As their display is handled in a more generic fashion than could be achieved by the plugin, these messages are typically used only for conveying unusual conditions to the user.

The one-to-many nature of DFC-to-DFD and plugin-to-application communications, which are implemented as a part of DFC-to-DFD communication, suggests the application of multicast IP sockets to facilitate message delivery. Each remote application is assigned a unique 8-bit identifier which is used to construct a multicast IP. Each DFD running the application uses the OS to add itself to the appropriate multicast group, causing it to automatically receive communications intended for that group.

## 1.3 Multicast IP

There are three addressing schemes available in IP: 1) unicast, 2) broadcast, and 3) multicast. Unicast is a one-to-one transmission and is the most common. Broadcast is a one-to-all transmission and is usually used in discovery protocols, such as DHCP. These packets are typically restricted to a single subnet to reduce overall traffic on the Internet. Multicast packets are transmitted from a single host to many hosts; in IP this can mean anywhere from zero to all hosts in a network. Like broadcast packets, multicast packets are typically limited to the local network, but they can be forwarded beyond the local network through the use of a multicast router. Multicast routers use the Internet Group Management Protocol (IGMP) to coordinate transmission of multicast packets between subnets [1].

IP multicast utilizes a group paradigm; when a host wants to receive multicast traffic, it joins the group associated with that traffic. Each multicast group has an associated IP address within the class D range (224.0.0.0 to 239.255.255.255). Sending traffic to a multicast group is accomplished by setting the packet's destination address to the group's IP. The Internet Assigned Numbers Authority (IANA) manages a list of IP addresses for multicast applications [2]. These IPs include 224.0.0.1, which is a group for all multicast-capable hosts, and 224.0.0.2, which is a group for all multicast-capable routers.

The one-to-many nature of the ISEAGE management network seemed to suggest that DeepFreeze take advantage of the benefits of multicast transmission. Because DeepFreeze manages communications channels for several applications, each application is given its own multicast group. DeepFreeze uses the 238.0.0.0/24 range; the IANA lists this range as reserved, so there should not be any conflicts.

Because TCP is limited to unicast connections, all packets using multicast addressing must use UDP. This is somewhat limiting, as DeepFreeze and the applications that execute within its environment must compensate for the lack of fault-tolerant features. Ideally, DeepFreeze would provide similar functionality—guaranteed, in-order delivery—so that this does not need to be repeatedly implemented by each application; current testing has not shown this to be necessary.

## 1.4 Wake-on-LAN

One of the much-needed features provided by DeepFreeze is power management. The process of powering up each computer and manually executing remote applications was an involved one, so for the majority of time the computers were simply left running. Besides being uneconomical, this compounded the need for a proper cooling solution before ISEAGE could run in its entirety. In order to power-down ISEAGE, the DFD running on each computer executes the system's *halt* program (this has several advantages over the *reboot* system call). The power-up procedure uses Wake-on-LAN (WOL) technology to signal the computer's motherboard to power on the system.

In order to activate the WOL functionality on the motherboard, a *Magic Packet* is sent to an attached network card (if there are multiple NICs, any should work). The payload of this packet is six bytes with value 0xFF, followed by sixteen repetitions of the target's MAC address, which is itself six bytes in length [3]. This packet must have a broadcast destination MAC address. Testing with the ISEAGE computers has shown that a UDP packet directed to port 7 and IP 255.255.255.255 will activate the WOL. Other combinations, such as the oft-suggested UDP port 9, were not successful.

## 1.5 File Descriptors and Local Sockets

One of the most prolific features of any Unix-like operating system is the philosophy that "everything is a file." Among other things, this means that any communications between a program and an opened terminal, FIFO, or socket behave exactly the same as with a file because the same system calls are used. Upon execution of a process, the OS allocates a table for files opened by a process; this is known as the file descriptor table. The first three entries in this table are of special significance and are known collectively as the standard streams, and individually as standard in (*stdin*), standard out (*stdout*), and standard error (*stderr*). Under conventional circumstances these are tied to a terminal where a user may send data to and receive data from the foreground program.

Most shell programs will allow file redirection, which allows any of these three descriptors to be redirected to a file, with the program proceeding to gather input from or produce output to this file. Output from one program may also be directly connected to the input of another through the use of a pipe. Both actions are transparent to the executing program because it treated terminal communication like file communication anyway.

DFD utilizes this feature to install itself as one of the layers of communication between the application and its plugin. It uses the *dup* family of system calls to replace the standard streams with a pair of anonymous local sockets allocated with the *socketpair* system call. Local sockets, also known as Unix-domain sockets, are similar to internet sockets in many ways, including the availability of both datagram and stream protocols, though they are restricted to providing inter-process communication (IPC) on the local machine [4: *socketpair*]. Figure 3 gives a visual representation of the communications channel created by a call to *socketpair*.

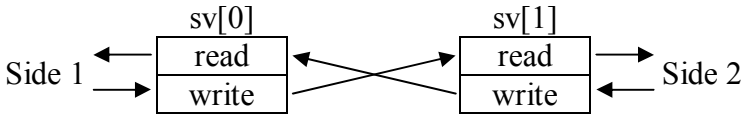


Figure 3: Anatomy of a *socketpair*

DFC uses local sockets to provide the backend for the plugin communications API. It also allocates a single local socket to asynchronously pass GUI messages, such as button clicks, to the DFC's underlying state machine. This allows for the use of a single *select* or *poll* system call to block execution until data is available or an event is pending. This is preferable to synchronization constructs due to simplicity, and busy-waiting due to efficiency.

## 1.6 XML, XML Schema, and the Xerces Parser

The configuration of DeepFreeze is complex enough to merit a discrete interface, but not so complex (or often-used) as to make developing a configuration GUI worthwhile. Extensible Markup Language (XML) is a web technology used to store information in a human-readable form with a unified syntax, making it fairly straightforward for a user to edit and a computer to parse. It has an inherently hierarchical format, making it useful in describing a number of complex patterns, including DeepFreeze's configuration.

The Apache Software Foundation releases an XML parser library called Xerces; it is available in several languages and is bundled with the JDK version 1.5, so it is widely used. It also provides support for a technology called XML Schema, which is an XML file describing the format of another XML file; Schema files also allow the use of regular expressions to restrict input format. If there is an XML syntax error or the user mistypes a tag, the parser will note the mistake and return a message describing the error. This is useful

because it allows developers to ignore syntactical mistakes and focus on the semantics pertaining to their specific model.

An example DeepFreeze configuration file is included in Appendix A. This file is used only by the DFC; any information needed by the DFDs is sent across the network.

## 1.7 GTK

The GIMP Toolkit (GTK+, or simply GTK) is a multi-platform API for GUI creation. Although written in C, its style is heavily object-oriented. As is standard among GUI APIs, the component drawing and event handling routines exist in one or more execution threads separate from the main thread. Because of this, it is important to provide proper synchronization for any shared data.

One of the standard GTK widgets is the GtkNotebook, which is a container that holds a number of tabbed windows. The DFC display window holds a single GtkNotebook; all management functionality provided by DeepFreeze is organized into a number of built-in notebook tabs. Additionally, the plugin API requires each plugin, upon initialization, to return a notebook tab containing all GUI elements. This modular organization gives plugin developers a great deal of autonomy in designing user interfaces.

## 1.8 Review

This chapter has introduced the DeepFreeze project and a number of the technologies incorporated into it. The architecture of ISEAGE was briefly discussed in order to provide insight both into the system that DeepFreeze is intended to control, and some of the design decisions behind DeepFreeze. Multicast networks, Wake-on-LAN, and local sockets were also introduced as DeepFreeze is highly dependent on these technologies. XML and GTK

were discussed, though these technologies were used out of preference and not out of necessity.

The next chapter will discuss related projects. The third chapter will focus largely on DeepFreeze implementation details and will expand greatly upon how the technologies discussed in this chapter are used. The final chapter will present conclusions and discuss the future direction of the project.

# CHAPTER 2. RELATED TECHNOLOGY

This chapter presents three technologies that provide functionality similar to a component of DeepFreeze. Also discussed are the advantages and disadvantages of each technology, and why its particular method or paradigm was not used directly within DeepFreeze.

## 2.1 inetd – The Internet Superserver

On Unix-like systems the *inetd* process is a central server that is responsible for handling the incoming connections of a specified list of Internet services [5]. It provides two features: 1) it reduces the number of processes in the system by having a single process wait for incoming connections, instead of using one such process for each service, and 2) the process handling the request may be functionally simpler because *inetd* handles details such as switching to the root directory and changing the process UID. While not a complete likeness, the DFD resembles *inetd* in many ways, most notably in the implementation of child process creation.

Upon execution, *inetd* reads its configuration file, which specifies services to listen for, the programs to handle service requests, and various other options associated with how the service handling programs should be initialized. For each service, *inetd* creates a socket of the appropriate type and uses the *bind* or *listen* system calls to link it with the local address associated with that service. It then calls *select* to wait for an incoming connection request. Upon receiving a request, a child process is forked. This child process proceeds to close all other file descriptors and uses *dup2* to copy a reference of the opened socket to the file

descriptors for the standard streams. The appropriate program is then executed. It performs all communication using the standard streams and exits upon completing the request.

While both *inetd* and the DFD require child processes to use the standard streams for communication, the rest of the DFD's process model differs significantly from *inetd*. Both open Internet domain sockets for their child processes, but the DFD does not give the child direct access to this socket, as it mediates the communications channel. Additionally, child processes for *inetd* terminate themselves upon handling the request, while child processes under the DFD are, by convention, designed to remain executing at all times (early termination is seen as abnormal).

## 2.2 Simple Network Management Protocol

A discussion of network management should bring to mind the Simple Network Management Protocol (SNMP). SNMP is an application-level protocol that uses UDP for datagram transport [6]. Within the SNMP model there are two types of network devices: 1) managers and 2) agents, which are devices being managed. It is designed to be device- and platform-independent, which allows a single manager to control many different types of agents.

At its basic level, SNMP defines a mapping of internal data or control variables to an external interface. These variables may reflect the state of an agent, or may be modified to control its behavior. Messages in SNMP can be divided into three basic types of communication: 1) the manager requesting a variable from an agent, 2) a manager controlling an agent by setting the value of a variable, and 3) the agent warning the manager of an abnormal situation. As part of standardizing the transfer of variables, SNMP defines several

data types. These include common types such as 32-bit integers and strings, and types that more closely resemble device behavior, such as a gauge variable, which is a value that is continually incremented until it reaches its maximum value, at which point it must be reset to continue counting.

Because SNMP runs on top of UDP, there has been discussion of using IP multicasting for distribution of management packets amongst naturally-occurring groups within the managed network [7]. The authors felt that while IP multicast protocols are somewhat restricting, there was advantage to using an existing protocol whose implementation is already part of the network stack used by the agents being managed.

While SNMP does define a standard interface for manager-agent communication, it was felt that the benefits of its use within DeepFreeze overweighed the cost of such an implementation. Additionally, the model behind DeepFreeze is dissimilar enough that it may diverge from the original intent of SNMP. DeepFreeze's DFC-DFD communications layer is best thought of a distributed state machine, while the manager-agent relationship is much more one-sided.

## 2.3 rdist – The Remote File Distribution Program

*rdist* is a program that aids in the distribution of files across multiple systems [8]. It uses a push methodology, where a central location may initiate the sending of a file to multiple systems. This contrasts with a pull methodology, where systems periodically poll a central location for a new version of the file. When transferring a file, it seeks to preserve the file's ownership, mode, and modification time.

Upon execution, *rdist* parses a *distfile*, which describes the target hosts and lists the files to be transferred. The *rcmd* function is used to contact an *rshd* server on each target machine and instruct it to execute the *rdistd* server. *rshd* uses standard stream manipulation to construct a communications tunnel between these two programs; once connected, the file transfer is negotiated.

Because *rsh* uses a set interface for host connection and stream construction, it is possible to tunnel the connection through another remote execution client/server suite. The SSH protocol is fully compatible with *rsh* syntax; this may be used to establish *rdist* connections securely.

While both *rdist* and the DFS application within DeepFreeze utilize standard stream manipulation to push files from a central location to multiple remote servers, the DFS differs in its use of multicast sockets to perform file transfers. Given certain scenarios, such as some combination of large files or a large number of target hosts, this can have significant advantages over multiple unicast connections. The ability for *rdist* to be tunneled through an SSH connection provides a security advantage over the DFS, though this is somewhat irrelevant as the DeepFreeze management network is designed to be self-contained.

When transferring files between hosts, the DFS seeks to preserve file ownership and mode. Unlike *rdist*, the DFS ignores modification time. Making such a feature meaningful would require the synchronization of system times, which slightly diverges in focus from the intended functionality.

# CHAPTER 3. IMPLEMENTATION

The chapter focuses on the implementation details of various components of DeepFreeze. It begins by discussing the reconfiguration of ISEAGE routers to support multicast routing, and continues with details concerning multicast communication within DeepFreeze. A section is devoted to each of the DeepFreeze Console and Daemon programs. Because the channel of communications between plugin and remote application is complicated, there is a section dedicated to reviewing and refining understanding of this channel. The chapter ends with sections that concern the implementation of two DeepFreeze applications.

## 3.1 Router Configuration

Multicast packets generally do not pass beyond the local network unless the router is configured to forward them. FreeBSD has build-in support for multicast networking at the host level, but additional steps must be taken to include support in a FreeBSD-based router. This configuration is necessary for the Icle file server computers, as the Icles run on their own subnet, with the file server acting as a router.

The first step is to compile multicast routing support into the kernel. Recompiling the kernel is fairly straightforward and is discussed in [9: 8]. The kernel configuration file must be modified to include the following line [9: 27.2.8]:

```
options MROUTING
```

The other procedure required to support multicast routing is the background execution of the *mrouted* system daemon. The following two commands will configure the FreeBSD server to run *mrouted*:

```
echo 'mrouted_enable="YES"' >> /etc/rc.conf
touch /etc/mrouted.conf
```

The first command appends a line to a FreeBSD configuration file informing the *rc* system to start *mrouted*. The second command creates an empty configuration file. *mrouted* requires the presence of its configuration file, even if empty.

While ISEAGE currently resides on a number of directly-connected subnets, there is the possibility that one or more components may be separated across the Internet to work with other organizations. This presents a problem as the intermediary routers will not forward DeepFreeze's multicast traffic. *mrouted* supports multicast tunnels, which are virtual connections between multicast routers on either end of a fully-unicast network. Any multicast traffic is forwarded through this tunnel, which is completely transparent to the multicast application.

## 3.2 Network Communication

DeepFreeze uses the 238.0.0.0/24 multicast network for all multicast traffic. When the DFD launches a remote application, it adds itself to a multicast group with the same last number as the application's ID (e.g., executing application 100 will cause DFD to listen to traffic intended for 238.0.0.100). The plugin communications API allows for both individual host communication and broadcast traffic, which causes a datagram to be sent to the application's multicast address. Application IDs 1 to 9 are reserved because their addresses are used for DeepFreeze-specific multicast groups. All communications between DFC and DFDs occur on UDP port 8.

All packets transmitted between DFC and DFD begin with a single byte that indicates the type of packet; these values are defined in the file *network.h*. Most of these different packet types are used for coordinating the state machines on the DFC and all controlled

DFDs; for example, there are several pairs of query-response packets that synchronize the state of a remote host and any remote applications it may be running. In-band and out-of-band application communications each have their own message type. The format of any data following the first byte is highly dependent on the packet type and will only be elaborated upon in the case of application packets.

Although IDs 1 to 9 are reserved, only 1 and 2 are currently used. The address 238.0.0.1 is used for all hosts running the DFC and the address 238.0.0.2 for all hosts running the DFD. These multicast groups are currently used for startup state synchronization, but may find other uses in the future.

One of the difficulties in configuring internet sockets for both the DFC and the DFD was selectively binding multicast sockets to a UDP port on a specific interface. The *bind* system call is fairly straightforward to use:

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons(DEEPFREEZE_UDP_PORT);
bind(sockfd, (struct sockaddr *) &addr, sizeof(addr));
```

This, however, will bind the socket to all network interfaces, which may be problematic on the ISEAGE computers, where the interfaces em1 and em2 are reserved for use by the Traffic Mapper. The solution is to bind to specific network interfaces by placing their IP in the *sin_addr* field of the allocated *struct sockaddr_in*. Surprisingly, binding to a multicast address is the same as binding to a specific interface, with the multicast IP used as the interface IP. As joining a multicast group requires specifying the desired network interface, the OS already ignores traffic on the other interfaces.

## 3.3 The DeepFreeze Console

The DFC is meant to be the central control interface for ISEAGE. Even so, there is support for multiple DFCs running in parallel so that the task of managing ISEAGE may be more easily divided. Further discussion of the inner workings of the DFC and its plugins is divided into a number of sections, each based on a natural division of the implementation.

### 3.3.1 Configuration File

The DFC uses the Apache Xerces library to parse its configuration file (see Appendix A for an example). Xerces was written to be used with a 16-bit character format, so there are some steps to convert between this and C's 8-bit format. Interpretation as a DeepFreeze configuration is accomplished in three steps:

1) Evaluate <hostspec> tags to get a list of the hosts the console will communicate with. This tag includes a *requires* attribute, which is used to specify another host that must be running before this host is sent the WOL signal. This is useful in at least two instances: 1) the host's file system resides on the required host and 2) the host is not on the same local network as the DFC, so the WOL signal must be relayed.

2) Evaluate <groupspec> tags to assign hosts to groups. These groups are temporary constructs used to make it easier to assign applications to hosts in the next step.

3) Evaluate <console> tags and the <app> tags within them; assign specified applications to running hosts. The <app> tag has three attributes: 1) application ID in the range 10-254, 2) name of the plugin on the local machine, and 3) name of the remote application on the remote machines.

DFC will evaluate multiple console specifications and choose one with a matching IP address. This allows multiple consoles to execute simultaneously on different computers, each running different applications. Any errors encountered during parsing are displayed in the message window (more on this below).

### 3.3.2 Host and Application Displays

The host and application displays are the active components of the command and control layer of the DeepFreeze Console (see Figure 4). Upon successfully parsing the configuration file, the DFC loads the host notebook tab and determines the state of each host. This displays a list of the hosts registered with this console and their corresponding statuses. The status may be one of the following values:

- Unknown – DFC is querying the state of this host
- Shutting down – The host is shutting down
- Down – The host is down
- Waiting on <IP> – The host is waiting for the computer <IP> to finish loading
- Waiting on dependents – The host is pausing to allow dependents to shut down
- Waiting to send WOL – The host is in the queue for WOL transmission
- Wake-On-LAN sent – A WOL packet has been sent and the computer is booting
- Up – The host is active

The On/Off buttons at the top allow the user to control the power state of the hosts. They are only active when all of the hosts are at a stable state (up or down), and then only the appropriate button is active. If a user chooses to power down the hosts, DFC will attempt to gracefully terminate any applications currently running.

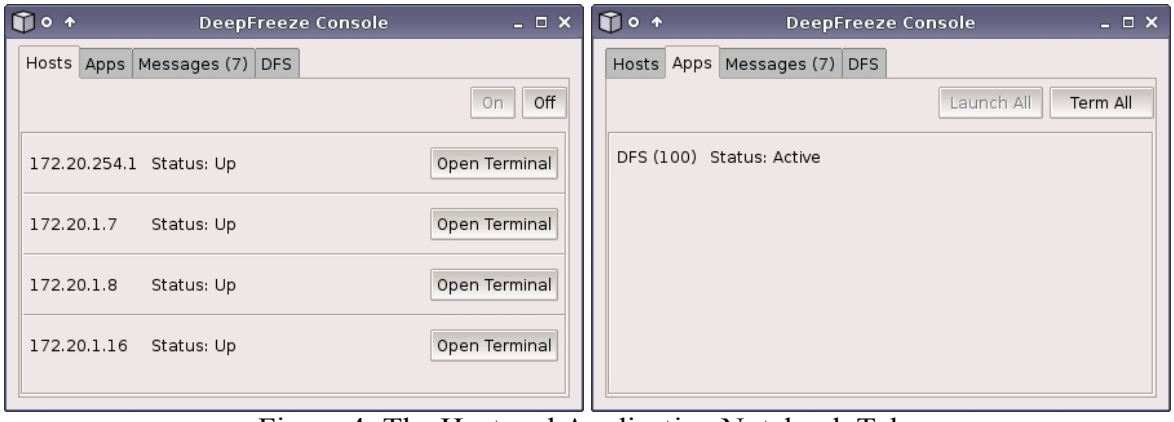


Figure 4: The Host and Application Notebook Tabs

Each host has a button that allows the user to open a remote shell. The environment variables are parsed to get the user's desired terminal software (e.g., *xterm*) and an SSH

connection is opened for the user *dfconsole*. This account should be available on all hosts, and should be a member of the *wheel* group so that the user may become *root*.

When all of the hosts are up, the application notebook tab is displayed; this notebook tab shows the name, ID, and status of each application. Upon loading the application notebook tab, the DFC queries each host for the status of its applications. The displayed status may be one of the following values:

- Querying – the DFC is querying the application's state
- Active – the application is running on all hosts
- Launching – the DFDs have been instructed to start the application
- Terminating – the DFDs have been instructed to terminate the application
- Partially active – the application is running on some hosts, but not all
- Inactive – the application is not running on any hosts
- Error – there was an error launching the application

Upon loading the application notebook tab, the DFC queries each host for the status of its applications. If any applications are fully active, then the DFC loads their associated plugin. The Launch All and Terminate All buttons are used to start and stop all of the applications controlled by the console. An application's plugin is unloaded prior to terminating its remote applications.

### 3.3.3 Wake-on-LAN Packets

In order to ensure that WOL packets are properly constructed, the functionality for generating such packets is included in both the DFC and the DFD. The DFC sends WOL packets to any hosts on its local network. Hosts not on the local network must have a "required host," which is specified in the configuration file. The DFC sends a relay command to the required host, which will then generate the WOL packet.

As WOL packets must have broadcast destination IP and Ethernet addresses, the output socket must be configured to support these features. This is accomplished using the *setsockopt* system call [4]. The *SO_BROADCAST* option allows the destination address to be a broadcast address; it is set to the broadcast address of the local subnet (as opposed to the global broadcast address 255.255.255.255). The *IP_ONESBCAST* option instructs the operating system to send broadcast packets out of the network interface associated with the socket [4: *ip(4)*]. The default behavior is to send packets out on the first available interface.

### 3.3.4 Message Display

The message display has two functions: 1) display messages from the DFC and any loaded plugins and 2) display out-of-band data from remote applications. Each message has a time, severity, and source associated with it. Messages are sorted according to arrival time. There are four severity levels:

1) Error – any fault that merits termination of the source
2) Warning – an error that does not interfere with execution
3) Info – information of use to the user
4) Debug – information of use to the application's developer

The list of messages may be filtered by severity and source using the appropriate buttons (see Figure 5). Displayed messages may be saved using the save button. The clear button deletes displayed messages. Both the save and clear buttons only apply to the set of messages that pass the message filter, all other messages are ignored.


Figure 5: The Message Notebook Tab

The message filters also act to prevent messages from being added to the list of messages; any message that arrives that does not pass the filter is ignored. Any unread messages are bolded, with the total number of unread messages appearing in the notebook tab's label.

The source of any out-of-band data received from a host is automatically set to the name of the transmitting application. The message is displayed in its original form, but with a message prefixed indicating the transmitting host and the fact that it is out-of-band data.

### 3.3.5 Compiling, Initializing, and Destroying Plugins

DeepFreeze plugins are compiled as shared libraries; these libraries are dynamically loaded at DFC startup and unloaded at DFC shutdown (using the *dlopen* and *dlclose* functions). Compilation of a shared library is different than a regular program [10]:

```
gcc –fPIC –c plugin.c –o plugin.o
ld –shared plugin.o –o plugin.so
```

The *fPIC* option enables position-independent code and the *shared* option instructs the compiler to continue without having all symbols defined.

Each plugin must define two functions:

```
extern "C" NotebookTab* plugin_initialize
    (struct in_addr h[], PluginSocket *ps, MessageInterface *mi);
extern "C" void plugin_destroy();
```

The *extern "C"* keywords are required in C++ code; this precludes the compiler from mangling the function name and preventing the symbols from being found. The *plugin_initialize* and *plugin_destroy* functions are called when the DFC wants to add or remove the plugin from the display (after the remote application starts and before it is terminated on all hosts). This may happen several times during a single execution of the DFC. Plugin initialization and destruction is different than plugin loading and unloading; the latter two happen only once during the DFC's lifetime.

The *plugin_initialize* function has three parameters. The first is a zero-terminated array of IP addresses of hosts that are running the associated remote application. This array is not guaranteed to remain valid throughout the plugin's lifetime, so the plugin must copy it if it intends to use it beyond the initialization procedure. The second and third parameters are discussed in the following two sections.

The initialization function returns a *NotebookTab*:

```
class NotebookTab {
public:
    GtkWidget* getPage();
    GtkWidget* getLabel();

protected:
    GtkWidget *tabPage, *tabLabel;
};
```

This is a wrapper class for two widgets: 1) the notebook tab's page and 2) the label. Most plugins will extend this class to add additional GUI components.

Most plugins will also want to create a separate execution thread to run in. If this is not done, the only time the plugin will receive control is during the initialization and destruction functions, and while handling a GUI event.

It should be noted that, due to the fact that they run as separate processes, there is some degree of separation between the DFD and its remote applications, while there is no

such separation between the DFC and its plugins. This means that an error in one plugin, such as a segmentation fault, can crash the DFC.

### 3.3.6 In-Band Plugin Communication

In-band plugin communication occurs through an object passed into a plugin upon initialization:

```
class PluginSocket {
public:
    virtual ssize_t send(const void *msg, size_t len, struct in_addr *to) = 0;
    virtual ssize_t recv(void *buf, size_t len, struct in_addr *from, int timeout) = 0;
    virtual int getRecvFD() = 0;
};
```

A call to the *send* method will create a packet out of the data in the *msg* buffer. The *to* parameter indicates the packet's destination. It may be either the defined value *INADDR_BROADCAST*, which will use multicast transmission to send it to all DFDs running the application, or the IP of an individual host (this IP is verified). In the latter case, the application's ID is added to the message to indicate the destination application; the former case does not require this as the application's ID determines its multicast address.

The *recv* method is designed to be a combination of the *recvfrom* and *poll* system calls. The *timeout* parameter is used to specify how long, in milliseconds, the socket should block waiting for data. If *timeout* is zero, the call will not block; if *INFTIM* the call blocks indefinitely. The method places up to *len* bytes of received data in *buf* and returns the size of this data (zero if *timeout* expired). The packet's source address is placed in *from*. If there is an error, the negative value of the corresponding error number is returned (see [4: *errno*]).

Although transparent to the developer, the DFC uses a local *socketpair* for each plugin to asynchronously cache data from the remote applications. When placing data in this buffer, the DFC prefixes the actual message with its source address; this is stripped out by the *recv* function.

The *getRecvFD* method returns the file descriptor for the read end of the *socketpair*. This allows the developer to include this descriptor in a *select* or *poll* system call.

### 3.3.7 Out-of-Band Plugin Communication

Out-of-band communication occurs through the other object passed in during plugin initialization, a *MessageInterface*. This class defines two methods:

```
enum MessageSeverity { ERROR = 0, WARNING, INFO, DEBUG, NUM_SEVERITY_TYPES };

class MessageInterface {
public:
    virtual void displayMessage(const char *src, MessageSeverity sev, const char *msg) = 0;
    virtual void vDisplayMessage(const char *src,
        MessageSeverity sev, const char *msgFormat, ...) = 0;
};
```

The *src* parameter defines a source for the message, which is displayed in the corresponding column in the message display interface (see Figure 5). This is typically set to the name of the plugin, but may be different if the developer wants to utilize multiple message sources. The *sev* parameter is set to one of the values in the *MessageSeverity* enumeration.

The *msg* parameter points to the text of the message. The *vDisplayMessage* method allows on-the-fly construction of the message using a *printf*-style message format, followed by a variable list of arguments. This message is constructed in a buffer with maximum size defined in *DISPLAY_MESSAGE_MAX_LENGTH*.

## 3.4 The DeepFreeze Daemon

The DFD is a server program that runs on each managed ISEAGE computer. It is responsible for executing, terminating, monitoring, and communicating with remote applications. This section begins with a description of setting up the DFD to run as a server program, and continues with implementation details, such as communications and remote application execution.

### 3.4.1 System Daemon Setup and FreeBSD Configuration

Because the DFD runs as a background system daemon, there are a number of additional procedures that must be followed to configure it as such and to make it conform to the BSD framework for server programs, called *rc*.

A program configures itself to run in the background by calling the *daemon* library function. This function performs three actions: 1) it forks off a child process, changes the child's session ID, and exits the parent process; 2) the working directory is changed to /; and 3) the first three file descriptors (usually the standard streams) are closed. At this point the program is completely disconnected from the terminal; any interaction with the program must occur using system signals, sockets, or some other form of inter-process communication.

The *rc* framework is used to standardize system daemon startup and shutdown procedures. Although not required, it has also been configured as a means of acquiring the state of a server program. In order to be included in the *rc* framework, the developer places an *rc* script in the */etc/rc.d* or */usr/local/etc/rc.d* directories. Upon system startup, these directories are searched and any scripts enabled in */etc/rc.conf* are activated. The script for the DFD (see Appendix B) is may be invoked with one of the following commands:

```
> /usr/local/etc/rc.d/dfd.sh start
> /usr/local/etc/rc.d/dfd.sh restart
> /usr/local/etc/rc.d/dfd.sh stop
```

The DFD requires a command-line parameter specifying the network interface it should use. Adding the following lines to */etc/rc.conf* will enable the DFD at startup and specify the appropriate interface:

```
dfd_enable="YES"
dfd_iface="em0"
```

Most daemon programs include a means of terminating a currently-executing copy of itself, usually through a command-line parameter. In order to support this, the DFD creates a *PID file* upon starting up. This file is located at */var/run/dfd.pid* and contains the process ID of the daemon. If this file already exists, the DFD will query the system to see if the process is running, if so it then exits without disturbing the already-running copy. If the *–kill* parameter is specified, the DFD sends the running process a *TERM* signal, which will terminate the daemon.

### 3.4.2 Process Execution

The DFD goes through a fairly complicated process to prepare communications channels for remote applications and to ensure that the applications are properly executed. The process begins with the DFD utilizing the *socketpair* system call twice to create two pairs of connected local sockets. One side of each of these will eventually replace the standard streams in the new process; the close-on-exec flag is set on the other side (this indicates to the OS that the socket should be closed when the *exec* system call is completed successfully). Next, a child process is split from the DFD using the *vfork* system call. Using *vfork* instead of *fork* provides two advantages: 1) reduced overhead due to the DFD's memory space being shared instead of copied and 2) the parent process is suspended until the child process either *exit*s or *exec*s [4]. The DFD exploits the second advantage to ensure that the process executed properly.

The child process proceeds by manipulating file descriptors, with the end goal being that one end of each of the two socketpairs is configured as a standard stream (see Figure 6). The first of the two socketpairs will be placed as *stdin* and *stdout*, and the second will be placed as *stderr* (the reasons for this are explained in the next section). Any local sockets

opened for other remote applications and any internet sockets (such as the socket denoted as MAIN, which is bound to the host's IP and is used to send all external traffic) already have their close-on-exec flags set. The DFD's end of the *stdin*/*stdout* socketpair is closed, but the *stderr* socketpair is left open in order to communicate a failure of the ensuing call to *exec*.

The child process then uses *dup2* to place the socketpairs in the range of the standard streams. However, this cannot be done directly, as it is possible that this would inadvertently close one of the sockets. Instead, *fcntl* is first used to duplicate the socketpairs to temporary file descriptors that are out of the range of the standard streams. After this, the old versions, which may or may not have been in the range of the standard streams, are closed. *dup2* may then be safely used to place the temporary descriptors back in the range of the standard streams. The temporary descriptors are subsequently closed.

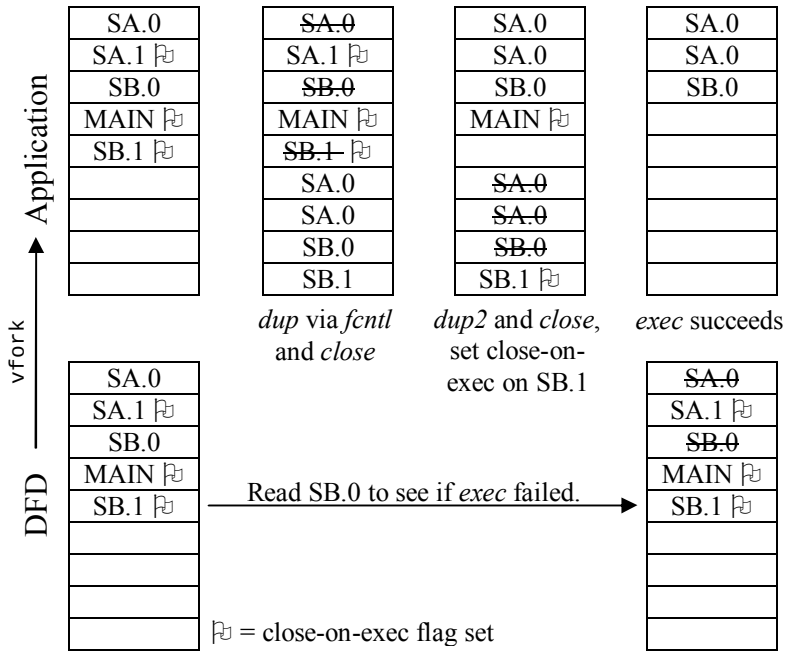


Figure 6: File Descriptor Manipulation throughout Remote Application Launch

At this point, the only extraneous descriptor that would remain open is the duplicate of the DFD's end of the *stderr* socketpair. As this is still needed, its close-on-exec flag is set to ensure that it is eventually closed.

The child process is ready to call *exec*. If the call is successful, the only file descriptors available to the remote application are the new standard streams. If the call fails (e.g., file not found, execute permission not set, invalid executable format, etc.), the MAIN socket descriptor is used to send an error message to the DFC and a single byte of data is sent over the DFD's end of the *stderr* socketpair. The child process then calls *_exit* (the *exit* call should not be used, see [4: *vfork*]).

When the child process calls *exec* or *_exit*, the parent process immediately returns from the call to *vfork*. Before closing the remote application's end of the two socketpairs, the DFD polls the remote application's *stderr* socketpair for data. If there is any data, then the parent knows the *exec* failed. It is safe to use the *stderr* socketpair in this manner because data only ever travels in one direction—from the remote application to the DFD—on this socket. Even though the byte of data indicating failure is conceptually traveling in the same direction, it is using a different pathway to do so, and so will not interfere with the normal communication channel.

Using the *stderr* socketpair as a means of detecting execution failure is an irregular solution. The more intuitive method would be to use the *waitpid* function with the *WNOHANG* option set (this prevents the process from blocking). Testing has shown that this solution does not work. It seems that even though the child process exits, the OS returns control to the parent before it marks the child as having terminated. Thus, a call to *waitpid* is unable to determine the child process's status.

### 3.4.3 Remote Application Communication

Like the DFC, the DFD provides both in-band and out-of-band communications channels to remote applications. Both types occur using the local sockets placed in the standard stream file descriptors during application launch. A remote application may receive in-band data from its plugin by reading *stdin* and may send in-band data to its plugin by writing to *stdout*.

Out-of-band data is sent to the user with a write to *stderr*. The first byte of data in the buffer must be a severity code, which may be obtained from the *MessageSeverity* enumeration introduced earlier.

Remote application communications have a maximum packet size of *MAX_APPLICATION_PACKET_SIZE*. This is calculated from the maximum Ethernet packet size, leaving room for some additional header information required by DeepFreeze's protocol.

### 3.4.4 Detecting Premature Application Termination

The DFD provides fault tolerance by determining when a remote application has terminated prematurely, reporting the nature of the termination, and automatically restarting the process. This is accomplished with a custom signal handler and the *wait4* system call.

When an application terminates, the OS sends a *CHLD* signal to the parent process. The process then enters a zombie state; this state's sole function is to hold the exit status of the process and wait for the parent to retrieve it. The parent does so with one of the *wait* family of system calls.

The DFD installs a signal handler for two signals: 1) *TERM*, which is used to terminate the DFD, and 2) *CHLD*. Upon receiving the *CHLD* signal, the signal handler uses a

local socket to notify the main execution thread that a child has exited. The main execution thread then retrieves child's exit status, which is one of three possibilities: 1) exited normally with the *exit* system call, 2) exited due to receiving a signal, and 3) exited from a signal with a core dump being generated (e.g., a segmentation fault). This status is reported to the DFC, as well as whether the remote application was successfully restarted.

### 3.4.5 Terminating Applications

The DFD terminates a remote application by sending it either a *TERM* or a *KILL* signal. The *TERM* signal is usually preferable as a process can catch this signal, do any necessary shutdown procedures, and terminate itself. A process cannot catch a *KILL*; it is terminated directly by the OS. If a process has not installed a *TERM* signal handler, then it uses the default handler, which terminates the application.

The actual signal sent is chosen by the DFC. As a convention, the DFC will instruct the DFD to send a *TERM* and then wait for an indication that the process closed. If it does not receive this indication within 10 seconds it instructs the DFD to send a *KILL*.

Intentionally terminated remote applications send a *CHLD* signal to the DFD, just as prematurely terminated applications do. In order to distinguish intentional from premature termination, a flag is set in one of the DFD's internal data structures that indicates the termination is intentional.

## 3.5 Review and Refinement of Communication Sequence

The channel of communication between a DeepFreeze plugin and its remote applications involves a chain of interacting components. Organizational constraints have placed the descriptions of these components in several places; this section provides a review

of the entire system in order to unify formerly disjoint routines. Additional details that were previously overlooked are discussed to ensure that this review is comprehensive. To provide a context for the review, consider a single challenge-response sequence from a plugin to its remote applications.

The plugin sends the challenge packet by calling the *send* method of the *PluginSocket* object passed in during initialization. The *send* method prefixes a header of up to two bytes to the packet before using the main internet socket (the socket bound to the host's IP on UDP port 8) to send the data across the network to the DFD(s). The first header byte is a packet type indicator that distinguishes it from command and control packets. The second byte is only used if the destination is an individual host; it is set to the application's ID. If the destination is the multicast network created for the application (i.e., the *to* parameter is *INADDR_BROADCAST*), then this information is not needed as it can be obtained from the last octet of the multicast IP.

Rather than assembling the packet in a separate buffer, which requires copying memory, the DFC utilizes a collection of I/O vectors to indicate that this data should be obtained on-the-fly (see [4: *read(2)*]). This is done by creating a number of *iovec* objects; each points to a memory location and the amount of data that should be gathered from that location. The *send* procedure creates two *iovec* objects, one for the header and one for the data. Changing the header size is as straightforward as changing the size indicator in the first object. The *iovec* objects are passed as parameters to a *sendmsg* system call.

The DFD receives the packet, examines the header, and determines that it is application data. It extracts the application ID from the second byte of the header if the packet is unicast and from the last octet of the destination IP if it is multicast. The ID is used

to retrieve the file descriptor for the other end of the remote application's *stdin* socket; the packet, minus the header, is written to this socket.

The remote application retrieves the packet with the *read* system call. It may use *select*, *poll*, or *read* to block until such data is available. Because the corresponding local socket uses a datagram protocol, each *read* is able to retrieve a single packet. As such, it is necessary to acquire the entire packet at once, rather than in parts as is possible with a stream protocol.

The remote application responds to the challenge packet by creating a response packet and sending it out using a *write* to its *stdout* file descriptor. The DFD reads this packet directly into its output buffer, leaving room for a two-byte header. As before, the first byte is a packet type indicator and the second byte is the application's ID. The second byte is no longer optional, as multiple remote applications may run on a single host. The DFD sends this packet to the DFC with a call to *sendto*.

The DFC extracts the application ID from the header and locates the corresponding plugin's *stdin* local socket. The local socket has no way of attaching out-of-band data, such as the packet's source address, so this must temporarily become part of the data stream. The *writev* system call, a *write* using I/O vectors, is used to prefix the IP address of the sending host to the packet, minus its two-byte header. When plugin calls the *recv* method of the *PluginSocket*, *readv*, an I/O vector version of *read*, is used to extract the IP address and packet data into the provided memory locations.

## 3.6 DeepFreeze's Distributed File System Application

The DFS is a concept application developed both to test the DeepFreeze API and to provide management functionality that will become useful as ISEAGE expands to multiple file servers. It has three functions: 1) browse and compare files and their attributes across multiple hosts, 2) modify file attributes such as ownership and mode, and 3) easily transfer files to multiple hosts. It should be noted that while the Icle computers were used to develop the application and generate the provided screenshot (see Figure 7), this application will most likely run on the file servers only.

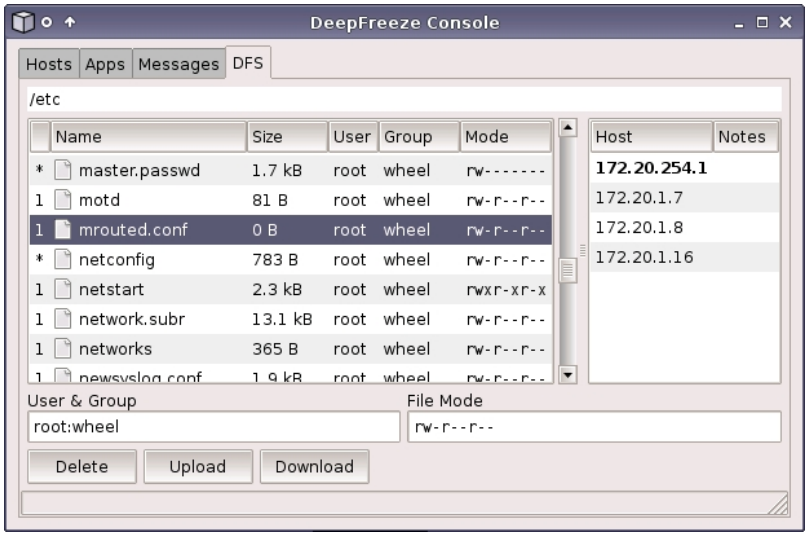


Figure 7: The DFS Notebook Tab

File system navigation is intended to mimic the behavior of any single-computer file browser. The requested display directory is shown in a read-only text box across the top of the window. The files are show in a list below this. To the left of the file name is a distribution indicator, which depicts the number of hosts with an equivalent file (an asterisk indicates all hosts). Files are considered equivalent across multiple hosts if their name, size, ownership, and mode (permissions) are the same. To see which hosts a file is active on, the

user may select a file and observe the host list; any hosts in bold text contain an equivalent file.

The user may browse to a directory that does not exist on all hosts. In this case, the hosts without the directory will be colored red in the host list. When changing to a new directory, the contents of all hosts containing a directory with that name are considered active, even if they are not considered equivalent due to size, ownership, or mode differences. The ownership and mode may be changed with the corresponding text boxes. User and group names are used instead of numeric IDs as the IDs may differ between systems.

File uploads take advantage of DeepFreeze's multicast architecture to reduce the number of packets needed to complete a transfer. The typical protocol model for a contiguous file transfer across the network is that the source sends a portion of the file and waits for the destination to indicate proper reception of that segment (in protocols that operate over TCP, this is a feature of the transport layer rather than the application layer). In the DFS the file segment is transmitted with a single multicast packet and the acknowledgement with multiple unicast packets. If a host does not acknowledge within a certain amount of time, then the packet is retransmitted, again with a multicast destination.

The MD5 hash of the file is calculated at both source and destination hosts using the algorithm included in the FreeBSD message digest library (see [4: *md5(3)*]). Upon completing the transfer, each receiving host sends the plugin its calculated hash; the user is alerted to any discrepancies. The calculation of the hash occurs as the file is being read from or written to the hard drive, rather than separately, in order to minimize disc activity.

## 3.7 The Traffic Mapper

The Traffic Mapper (TM) is the quintessential DeepFreeze application, though the first version of the TM existed before DeepFreeze. DeepFreeze was designed to alleviate the difficulties encountered with administration of the TM. Prior to DeepFreeze, the TM was started from a central workstation connected to the host computers through multiple SSH sessions. The TM would display raw information about the packets it sent and received. This information was only useful during debugging, although one could determine relative network activity by examining how fast the text was scrolling in the SSH window. Periodically, one of the TM processes would crash. This was disruptive to events such as the Cyber Defense Competition (CDC), where the error would usually go unnoticed until one of the teams complained that their services were all running, but could not be accessed from an external subnet.

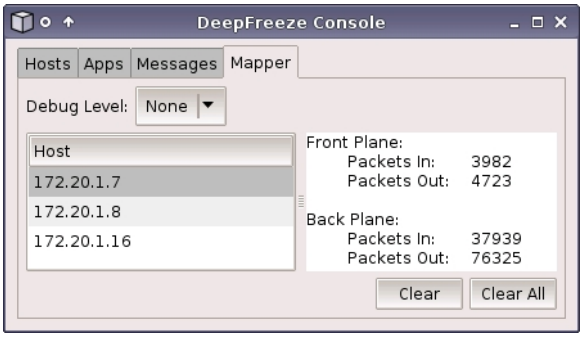


Figure 8: The Traffic Mapper Notebook Tab

The original TM had to be modified in order to make it compatible with DeepFreeze. The majority of the modifications involved migration to DeepFreeze's system of out-of-band communication. There were two additional features added to merit the need for GUI components (see Figure 8): 1) on-the-fly modification of the debug level, which was previously set with a command-line parameter, and 2) collection of elementary statistics in

regards to the front plane and back plane network interfaces on the host (em2 and em1, respectively, in Figure 1). The TM simulates a certain number of routers within each host; future modification will include statistics displays for each of these routers.

One of the projects ISEAGE has been tasked with is the simulation of Iowa's network, known as the Map Iowa Project. This project is a Homeland Security scenario involving a physical or cyber attack on Iowa's network, with the focus being on how the network reacts to certain routers or lines of communication becoming unavailable. The end goal of the control interface is the real-time addition and subtraction of network nodes and links. Both the TM remote application and plugin are going to have to be modified to allow for such an event.

# 4. CONCLUSIONS

DeepFreeze utilizes a wide variety of technologies to provide functionality and make its user and programming interfaces intuitive. This thesis has discussed the implementation of DeepFreeze as both a command and control interface and as an application development platform for the ISEAGE project. Two applications that were developed or modified to operate within the DeepFreeze environment were also introduced.

## 4.1 Future Work

This version of DeepFreeze is considered a viable release. It has been extensively tested and many issues involving corner cases have been resolved. While it still remains to be used in a live environment, such as the CDC, this promises to be a successful endeavor. Future iterations of DeepFreeze may incorporate additional features for increased flexibility in the way applications interact within the ISEAGE environment. This section will present several ideas for future modifications.

A highly desirable feature, and the most pressing, is the inclusion of packet transmission assurances, namely guaranteed and in-order delivery, for the communications channels between plugins and their remote applications. This would centralize fault-tolerant communications instead of forcing each application to provide its own. Although the TCP protocol provides this for stream sockets, DeepFreeze's multicast model makes the use of these sockets impossible. Even so, the implementation of this feature may end up being functionally similar to the TCP state machine.

Although it falls outside of DeepFreeze's original model, it may become desirable to allow plugins to communicate with each other. Two communications channels come to mind:

local sockets and shared memory. Creating local sockets between plugins would only require slight modification of the API and the configuration file. Local sockets would also allow for easier integration with presently utilized design patterns. At the same time, shared memory may allow for a more natural and efficient communications channel, but API modifications are likely to be more extensive to support this feature. A similar feature may also become desirable for remote applications.

The current version of DeepFreeze expects hosts to utilize the built-in power management features. Placing hosts on a different subnet requires a chain of hosts between the DFC and the subnet with the hosts. This is not an issue with the current configuration of ISEAGE. However, future projects may require hosts deployed on non-adjacent subnets, such as the case where DeepFreeze components need to be placed across the Internet. It is recommended that such a project utilize a VPN to place the remote host on a virtual, adjacent subnet. An alternative is to allow hosts to operate outside of the power management system.

Finally, although many applications do not fit the DeepFreeze one-to-many model, it is still desirable to incorporate them within an all-inclusive management interface. The DFC could be modified to allow plugins without respective remote applications, thereby increasing the variety of applications that may operate within DeepFreeze's management console.

While these features would enhance DeepFreeze for specific uses, they were not included in the current inception of DeepFreeze because they are outside the focus of the project. It is suggested that as developers find a need for additional functionality, they modify DeepFreeze to incorporate it.

# APPENDIX A: EXAMPLE DFC CONFIGURATION FILE

```
<?xml version="1.0" encoding="utf-8" ?>

<distribution xmlns="http://www.iseage.org/DeepFreeze">
    <console ip="172.20.254.212">
        <app name="DFS" id="100"
               interface="/usr/local/DeepFreeze/dfs.so"
               exename="/usr/local/sbin/dfs">
            <group name="iclesFS" />
        </app>

        <app name="Mapper" id="200"
        interface="/usr/local/DeepFreeze/mapper.so"
        exename="/usr/local/sbin/mapper">
            <group name="icles" />
        </app>

        <!-- Can also use a host tag to specify hosts running the app. -->
    </console>

    <!-- Icle #1 -->
    <hostspec ip="172.20.254.1" mac="00:30:48:73:b3:ec" /> <!-- File server -->
    <hostspec ip="172.20.1.1"   mac="00:0e:0c:65:eb:71" requires="172.20.254.1" />
    <hostspec ip="172.20.1.2"   mac="00:0e:0c:65:e7:dc" requires="172.20.254.1" />
    <hostspec ip="172.20.1.3"   mac="00:0e:0c:65:eb:74" requires="172.20.254.1" />
    <hostspec ip="172.20.1.4"   mac="00:0e:0c:65:eb:76" requires="172.20.254.1" />
    <hostspec ip="172.20.1.5"   mac="00:0e:0c:65:eb:8b" requires="172.20.254.1" />
    <hostspec ip="172.20.1.6"   mac="00:0e:0c:65:eb:8f" requires="172.20.254.1" />
    <hostspec ip="172.20.1.7"   mac="00:0e:0c:65:eb:d0" requires="172.20.254.1" />
    <hostspec ip="172.20.1.8"   mac="00:0e:0c:65:eb:8a" requires="172.20.254.1" />
    <hostspec ip="172.20.1.9"   mac="00:0e:0c:65:1c:d8" requires="172.20.254.1" />
    <hostspec ip="172.20.1.10"  mac="00:0e:0c:65:1b:fa" requires="172.20.254.1" />
    <hostspec ip="172.20.1.11"  mac="00:0e:0c:65:1e:69" requires="172.20.254.1" />
    <hostspec ip="172.20.1.12"  mac="00:0e:0c:65:1c:d9" requires="172.20.254.1" />
    <hostspec ip="172.20.1.13"  mac="00:0e:0c:65:e8:a8" requires="172.20.254.1" />
    <hostspec ip="172.20.1.14"  mac="00:0e:0c:65:e8:a3" requires="172.20.254.1" />
    <hostspec ip="172.20.1.15"  mac="00:0e:0c:65:eb:7b" requires="172.20.254.1" />
    <hostspec ip="172.20.1.16"  mac="00:0e:0c:65:eb:43" requires="172.20.254.1" />

    <groupspec name="icles">
        <subnet range="172.20.1.0/27" />
        <!-- <subnet range="172.20.2.0/27" /> -->
        <!-- <subnet range="172.20.3.0/27" /> -->
        <!-- <subnet range="172.20.4.0/27" /> -->
    </groupspec>

    <groupspec name="iclesFS">
        <host ip="172.20.254.1" />
        <!-- <host ip="172.20.254.2" /> -->
        <!-- <host ip="172.20.254.3" /> -->
        <!-- <host ip="172.20.254.4" /> -->
    </groupspec>
</distribution>
```

# APPENDIX B: DFD RC SCRIPT (DFD.SH)

```
#!/bin/sh
#
# PROVIDE: dfd
# REQUIRE: DAEMON
# BEFORE:  LOGIN
# KEYWORD: shutdown
PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin

. /etc/rc.subr

name="dfd"
rcvar=`set_rcvar`

command="/usr/local/sbin/${name}"

pidfile="/var/run/${name}.pid"

start_cmd="${name}_start"
restart_cmd="${name}_restart"
stop_cmd="${name}_stop"

load_rc_config $name
eval "${rcvar}=\${${rcvar}:-'NO'}"

dfd_start() {
        ${command} ${dfd_iface}
}

dfd_restart() {
        dfd_stop;
        dfd_start;
}

dfd_stop() {
        check_pidfile ${pidfile} {$name};
        ${command} -kill
}

run_rc_command "$1"
```

# BIBLIOGRAPHY

[1] *Internet Group Management Protocol*, Version 3. RFC 3376. 2002, Oct.

[2] IANA. (2006, Aug.). Internet Multicast Addresses. Internet Assigned Numbers Authority. [Online]. Available: http://www.iana.org/assignments/multicast-addresses

[3] Wake-on-LAN. (2006, Dec.). Wikipedia. Wikimedia Foundation, St. Petersburg, FL. [Online]. Available: http://en.wikipedia.org/wiki/Wake-on-LAN

[4] MAN. FreeBSD Hypertext Man Pages. The FreeBSD Project. [Online]. Available: http://www.freebsd.org/cgi/man.cgi

[5] W. R. Stevens, *UNIX Network Programming*, 1st ed. Englewood Cliffs, NJ: Prentice Hall, 1990, pp. 334-339.

[6] B. A. Forouzan, "Network Management: SNMP," *TCP/IP Protocol Suite,* 3rd ed. New York: McGraw Hill, 2006, ch. 21.

[7] J. Schönwälder, "Using Multicast-SNMP to Coordinate Distributed Management Agents," in *Proc. of IEEE 2nd International Workshop on Systems Management*, 1996, pp. 136-141.

[8] M. A. Cooper, "Overhauling Rdist for the '90s," in *USENIX Systems Administration (LISA VI) Conf.*, Long Beach, CA, 1993, pp. 175-188. [Online]. Available: http://www.magnicomp.com/download/rdist/overhaul.ps.gz

[9] Handbook. FreeBSD Handbook. The FreeBSD Project. [Online]. Available: http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/

[10] D. A. Wheeler. (2003, Apr.). Program Library HOWTO. [Online]. Available: http://www.faqs.org/docs/Linux-HOWTO/Program-Library-HOWTO.html

J.-M. de Goyeneche. (1998, Mar.). Multicast Programming. [Online]. Available: http://tldp.org/HOWTO/Multicast-HOWTO-6.html

# ACKNOWLEDGEMENTS

I would like to thank my committee members for their support both in this project and in my graduate academic career. Special thanks to Drs. Jacobson and Daniels, who have helped me grow my ideas into reality.

To my family: words cannot express my gratitude for all you have given me. I have worked hard to become a good person in the hope that someday I can provide for others the loving support you have provided me.