



All About CRAM: The CREATE Real-time Application Manager

The CREATE Real-time Applications Manager (CRAM) is a framework for developing, deploying, and managing distributed real-time software. It has evolved in our group at UCSB through three implementations over the space of five years. The background of CRAM is the work done since the early 1990s on distributed processing environments (DPEs), which started in the telecommunications industry (see Appendix 1). CRAM is unusual among DPEs in that it is very light-weight and efficient, but also fault-tolerant, and that it supports both planning-time and run-time load balancing as required by real-time applications. Its main application areas to date are large-scale music performance systems and distributed virtual environments.

Introductory Scenario

Distributed applications are software programs where the components (service programs) run on different computers connected via a network. Often, this corresponds to the client/server design pattern.

As an initial scenario to motivate the need for distributed real-time systems, imagine a multi-computer music performance application where one computer is reading data from one or more input devices (MIDI controllers, head trackers, computer vision systems, etc.) and mapping this controller data onto concrete parameters for a given set of synthesis programs. We can call this program the input server. In our scenario, another computer is running a synthesis server, which takes commands from the input server, and performs software sound synthesis, sending its output over the network to some output server. The output server is a program that reads sound sample blocks coming in via the network from the synthesis server and mixes and spatializes them, sending its output to a multichannel sound output interface.

The problem now arises of how we are to start and stop this application in a controlled manner. One could do it manually; logging in to the various machines to start the output server, then the synthesis server, then the input server. To better manage distributed applications, one needs software tools that can remotely start, stop, and monitor software on several computers connected by a network. This is the task of distributed processing environments (DPEs).

A somewhat more complex multi-server CRAM configuration is illustrated in Figure 1 below. In this example, each of the round-edged rectangles is

a separate server program. The top four servers are synthesis and processing programs (written in C++ using the CSL framework [Pope and Ramakrishnan, 2003]); the larger box in the middle is the CRAM system manager (written in Smalltalk), and the control server at the bottom (also written in Smalltalk using the Siren framework [Pope 1986, 2003]) sends messages to the CSL servers to trigger the sound synthesis.

The control links (shown as dotted lines) use the CRAM and OSC (OpenSoundControl [Freed and Wright, 1997]) protocols, and the inter-program sample streams (drawn as arrows) use the CSL sample streaming protocol. CRAM manages the distributed application; it starts up the services, and monitors them during run-time.

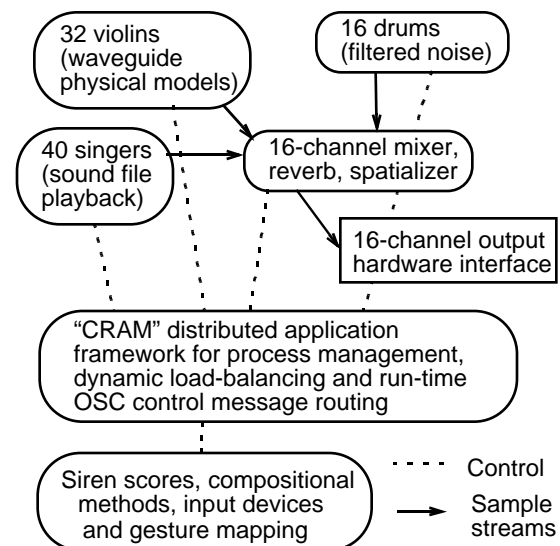


Figure 1: A distributed CRAM configuration (each box is a separate server)

The CRAM Model

The CRAM data model comprises hardware and software; hardware is represented as networks of nodes (computers), and software is seen as applications that are decomposed into individual services. The CRAM databases and tools (see below) reflect this data model. Figure 2 below shows the relationships of the components of a CRAM system. At run-time, the system manages services running on nodes; the abstractions of networks and applications are used for planning and management.

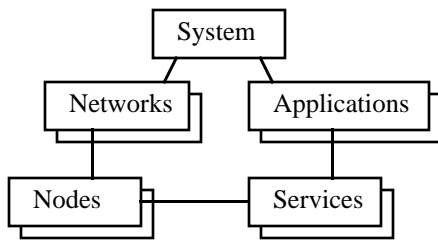


Figure 2: The CRAM Data Model (connecting lines represent composition or “has-a” relationships)

The Parts of a DPE

A traditional distributed processing environment (a DPE) generally consists of at least three components: a node manager, a service interface, and a system manager.

The node manager is a simple daemon (a stand-alone background program) that is assumed to be running on each computer that the DPE intends to manage. Node managers accept commands from the system manager to start/stop/monitor remote services. Their simplest role is as a “remote execution” service, though they also support status monitoring, fault-recovery, and load balancing.

A DPE service interface is a set of functions that applications need to implement in order to be managed by a DPE. This functionality is normally packaged as a library class that a developer includes in an application. The service interface functions include basic start/stop methods, and some sort of status query; they are used by the node manager to control the service.

The third component is the system manager; it uses node managers to start the components of a distributed application. DPE systems often use databases to describe network hardware facilities and the composition of services into applications. Advanced system managers also included off-line

application planning tools that perform the resource allocation for deploying an application on a specific network.

In CRAM, the node manager is a small program written in C++ that uses a simple socket-based protocol to talk to its services. The system manager uses the same protocol to communicate with node managers. The service interface component that is incorporated into application programs includes code that implements this protocol, and starts a “listener” thread when the application starts. This thread waits for commands from the node manager to control the application.

There are two features that are often considered optional in DPEs, but are central to CRAM: fault-tolerance and load-balancing. For applications that require robust software (e.g., music performance), the system must be able to identify and recover from a hardware or software fault within a small number of seconds. For large-scale systems that are to handle dynamic processing and I/O loads (e.g., music performance), some manner of planning-time as well as run-time load-balancing is also necessary. We will discuss these features more below.

CRAM in Action

To start and manage a distributed application using CRAM, we first assume that the network nodes (computers) are known, and that we have access to a database with information about them. We also assume that node manager daemon programs are running on the nodes of the network. Lastly, we assume that the software we want to use is installed on the computers, or on a file server to which the network nodes have access.

We describe applications as collections of services running on nodes. A service is just an application program, written in any arbitrary language, that implements the CRAM service interface functions (including the command listener thread). The three services that we introduced in the scenario above (input, synthesis, and output) are candidate CRAM services.

To manage the example application using CRAM, we need to store a description of it in the CRAM database. This simply means that we define which service is to run on which node. If we have computers named waltz, jerk, and belly (all of our computers are named after dances), then we would send the following SQL (structured query language) command to the database:

```

insert into applications (name, services)
values ('SirenCSL',
{
  "jerk.input_server",
  "belly.synthesis_server",
  "waltz.output_server" } );

```

Another table in the database describes service types, which include the service name as well as its requirements (e.g., the output server must run on a node with multichannel output hardware) and run-time options. Once this is done, we can start up the CRAM system manager. When it starts, it loads the database tables that describe nodes on the network, types of services and their options, and applications. We can use the system manager to make sure that the application will run (i.e., that the nodes are all on and have node managers running on them), and then tell it to start the application. When we do this, the system manager sends messages to each of the node managers requesting that they create and initialize the services described in the database. Once these are all ready, the application is started by sending “start” messages to the services.

While the application is running, we can use the system manager to monitor its status. The default two-pane system manager view as shown in Figure 3 below.

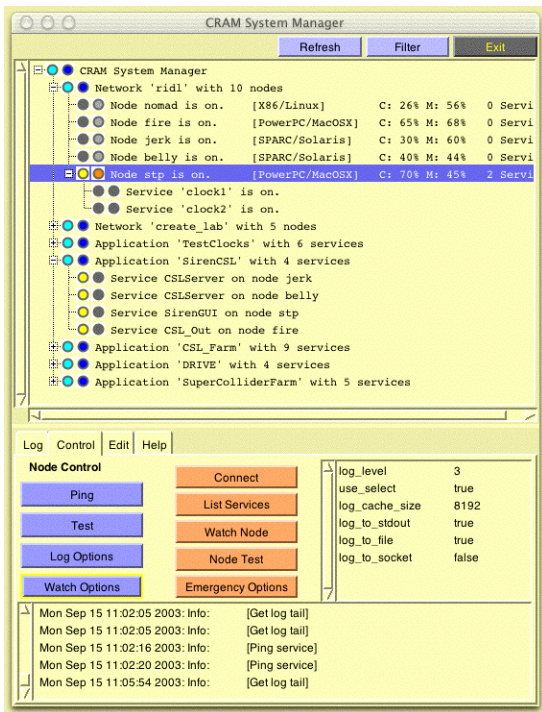


Figure 3: The CRAM System Manager GUI
The upper pane of the window is the CRAM

tree view; the top-level items in the tree are the lists of known networks and applications. The network branches expand into lists of nodes on a given network (as visible in the Figure), and nodes expand to show the services running on them. An application branch expands into the services that comprise the application, which may be running or not.

Below this pane is the log/control/edit/help pane, a tabbed view that generally shows the output log of the selected tree item, or, in control mode as shown in the Figure, can be used to operate on the selected CRAM object (e.g., connect to a node or start a service). In the Figure above, a node is selected, and the control view shows the operations that are possible (in the upper left of the lower view of the Figure). The table to the right of the buttons displays the node’s manager options, and the text field below this gives the recent log messages from the node manager.

CRAM Object Lifecycles

CRAM Service

A service can be any program that is used by a CRAM application. Services are assumed to be unreliable; they come and go. It is CRAM’s job to keep a certain number of them alive for a certain period of time.

The stages in the start-up of a service are create, initialize, and start; once a service is running, the node manager can “ping” it, send it test messages, or get/set run-time options. Services also provide suspend and resume commands, and a stop (kill) command. Advanced features that increase the fault-tolerance of applications include watch options and heartbeats (described in more detail below).

CRAM Node Manager

Each computer on a network is assumed to run a CRAM node manager daemon program. It’s a very bad thing for these to crash. Generally, the node manager is started from some low-level script (e.g., /etc/rc.local), and special measures are taken for these programs to be stable, network accessible, and insensitive to other failures. The CRAM system manager has several techniques for locating the node manager on a given computer.

If a node manager “hangs” (i.e., stops responding to commands sent to its listener port), a special facility uses a second command port (the “emer-

gency” port) to kill and restart the main listener thread.

CRAM System Manager

The CRAM system manager is the management tool and GUI for supervising CRAM applications. One can stop and restart the manager while an application is running—the manager’s first job is to find out what the current environment is (i.e., discover node managers on the selected network and list their services).

CRAM Application

A CRAM application is simply a collection of services running on a given network. It is the job of the system manager to start and monitor applications. Since services are assumed to be untrustworthy, the system manager is given the job of restarting them when necessary in order to keep the application running.

CRAM System Manager

The CRAM system manager is a Smalltalk application; to start it, use the shell script start-CRAM that’s usually installed in /usr/local/CRAM/bin. It takes a few seconds to start up and load its database tables. Once it’s running, you will see the manager view shown in Figure 3 above.

Tree pop-up menus

Log view

Control views for nodes and services

Writing CRAM Applications

To add CRAM-compatibility to an application, a developer needs to incorporate a class derived from the library class Server; this is the server interface. The simplest example is a service that logs the time at regular intervals—the clock service. the C++ header for this application is given below.

```
// The Clock Service class

class ClockService : public Server {

protected: // protected members
    void start(void); // start method
    void stop(void); // stop method

    // suspend/resume are optional
    void suspend(void); // suspend method
```

```
void resume(void); // resume method

// options initialization/setter methods
void init_options(); // initialize my options
// set an option value
void set_opt(char * nam, char * val);
// application-specific state
pthread_t mLoop; // clock thread

public:
    unsigned mPeriod; // the clock’s periods
    bool mRunning; // am I running?

// Constructor/destructor
    ClockService(char * name,
                 int lport, int eport,
                 unsigned period);
    ~ClockService();
};
```

CRAM database tables

Networks

Nodes

Service types

Applications

Installing CRAM on a network

Database

NodeManagers

Fault-tolerance in CRAM

Service hang-ups

NodeManager death/restart/service discovery

System Manager death/restart/node discovery

Implementation Notes

A very simple packet protocol spoken between the system manager and a node manager, based on connection-less UDP sockets

Safety and exception handling

Platform dependencies

POSIX pthreads package (pthread_kill())

select system call

Unix-style file I/O

Log and PID files and names

setsockopt arguments (Solaris)
var_args and stdio

References

- S. Bannerman, B. Stockdell, and M. Stutz, "Network Topology Configuration Management Experience Report," *Proc. TINA '99*, Turtle Bay, Hawaii, pp. 137-39.
- Freed, Adrian and Matthew Wright. 1997. "Open SoundControl: A New Protocol for Communicating with Sound Synthesizers." *Proc. ICMC 1997*.
- McCartney, James. 1996. "SuperCollider: a new real time synthesis language." *Proc. 1998 ICMC*.
- Pope, S. T. 1986. "The Development of an Intelligent Composer's Assistant: Interactive Graphics Tools and Knowledge Representation for Composers." *Proc. 1986 ICMC*.
- Pope, S. T. 2001. "Music and Sound Processing in Squeak Using Siren." in Guzdial, Mark and Kim Rose. *Squeak: Open Personal Computing and Multimedia*. (book and CD-ROM) Prentice-Hall.
- Pope, S. T., A. Engberg, F. Holm, and A. Wolf. 2001. "The Distributed Processing Environment for High-Performance Distributed Multimedia Applications." *Proc. 2001 IEEE Multimedia Technology and Applications Conf.*, U. C. Irvine.
- Pope, S. T. and C. Ramakrishnan, 2003. "The CREATE Signal Library ("Sizzle"): Design, Issues, and Applications." *Proc. 2003 ICMC*.

Appendix 1: DPE Background

During the 1990s, distributed processing environments were developed in the telecommunications industry to run large ATM networks. These efforts were based on a data model called the information network architecture or INA.

The first set of INA specifications, known as the Cycle 1 Specifications was released by Bellcore (a branch of Bell Labs) in June 1992, with a revised version being released in April 1993. An international research initiative set up in 1993 and known as the telecommunications information network architecture consortium (TINA-C) has developed and extended the INA framework. TINA-C defines

a DPE with many of the features of CRAM.

- <http://www.tinac.com>
<http://hegel.ittc.ukans.edu/projects/tina-c>
<http://www.nm.informatik.uni-muenchen.de/Vorlesungen/ws9900/tks/teil15.pdf>
<http://www.hpl.hp.com/hpjournal/96oct/oct96a2.pdf>
<http://www.item.ntnu.no/fag/SIE5050/HTML/pensum/Distributed%20Processing%20Platform.doc>
<http://www.comsoc.org/livepubs/surveys/public/1q00issue/berndt.html>

Appendix 2: How is CRAM different from traditional INA-based DPEs?