DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Relatório Técnico

RT-MAC-9516

Supporting Distributed Application Management in Sampa

Markus Endler
Anil D'Souza

Novembro 95

Supporting Distributed Application Management in Sampa

Markus Endler *and* Anil D'Souza
Departamento de Ciência da Computação
IME-Universidade de São Paulo
São Paulo, Brazil
E-mail: {endler,anil}@ime.usp.br

1 Introduction

Distributed applications are becoming bigger, more complex and increasingly dependent on other distributed programs and services. This situation creates an higher demand of systems that support on-line management of distributed programs and services through monitoring and dynamic configuration facilities. Such management may be used for guaranteeing the reliable and efficient execution of distributed programs, and the availability of essential services or processes in spite of node or communication failures.

Compared to the traditional network management, *Distributed application management*[7] handles runtime information at a higher level which is more related to the application domain than to the basic data units exchanged between the network components. Distributed application management in general, consists of configuring, monitoring and controlling application processes and servers and the communication links (bindings) between these processes. It may be oriented towards different purposes, such as fault, performance, security or configuration management. This work is concerned with distributed application management focused on the automatic enforcement of availability policies.

Some work has been done in implementing specific tools for monitoring and controlling distributed applications, such as the Meta Toolkit[12], the Megascope tool[15] within the Project Pilgrim[14] and the tools developed by Huang and Kintala[8]. However, until now, there is no system that supports the management of DCE-based applications with respect to fault-tolerance and availability requirements.

Sampa, which stands for *System for Availability Management of Process-based Applications*, has been designed to support the management of fault-tolerant DCE-based services and programs through the enforcement of application-specific availability specifications. It will be a decentralized and fault-tolerant system based on several DCE services, such as RPC, the Cell Directory Service (CDS) and the Distributed File System (DFS).

Sampa is supposed to detect faults such as node crashes, network partitions, process crashes and hang-ups, and to automatically execute the necessary recovery actions according to a user-provided availability specification. Due to Sampa's limited support for checkpointing its fault detection and recovery capabilities are constrained to the automatic detection & recovery of the faults mentioned above, and periodic checkpointing and recovery of

some of the program's internal state, which corresponds to level 2 in Huang&Kintala's[8] classification of fault-tolerance facilities. Therefore, the main application areas for such kind of fault tolerance are systems with higher demands on availability than on strong data consistency, such as telephone switching systems or information retrieval systems.

In this paper we will focus on the design of the base services within Sampa and show how they can be used for managing a generic replicated service. After presenting the system's global architecture and its main components in section 2, in sections 3, 4 and 5 we describe the design and main features of its reliable group communication, monitoring and checkpointing facilities, respectively. Then, in section 6 we present Sampa's language for writing availability specifications along an example. In sections 7 and 8 we then comment on related work and draw some conclusions on the current status and future steps of the project.

2 Sampa's Global Architecture

Sampa's architecture is based on the principle of separating all sorts of management functions, (e.g. monitoring, process management, checkpointing) into two levels. At a lower level, *agents* executing on every host execute all sorts of local process management tasks (e.g. process creation and deletion, checkpointing) and monitoring operations (e.g. collection, filtering and data pre-analysis) and interact with a supervisor process for notifying fault occurrences and receiving monitoring and reconfiguration commands.

At a higher level a *supervisor* takes global management decisions for each distributed service (or application program) according to its availability specification and based on the monitored data received from the agents. The availability specification includes global monitoring and reconfiguration directives that are interpreted by the supervisor and are further delegated as simpler commands to the corresponding agents. The supervisor also provides a user interface for configuring the distributed programs/services in an ad-hoc manner.

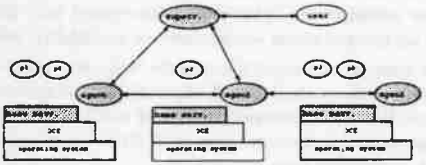

Figure 1: The Sampa Architecture

Following this approach, Sampa's architecture (Figure 1) consists of a set of *base services* and a set of *agents* available on every system node. The agents are responsible for monitoring several hardware, operating system and DCE resources at their node (e.g. CPU and memory utilization, RPC request rate, etc.) and analyzing the data before sending it to the supervisor. Besides this, the agents have full control of all application processes and servers executing on their node. Due to the fault-tolerance requirements of the system itself, agents are also in charge of monitoring the availability of the other agents and of reporting any failures (node crash, network partition) to the other agents and to the supervisor.

Although in principle this architecture may be used with more than one supervisor, each of which specialized in a certain kind of management (e.g. performance, configuration or availability management) our initial goal in the project is to implement a single supervisor which controls monitoring, system availability and reconfiguration.

2.1 Base Services

For implementing fault tolerant services and applications, a set of base services is currently being developed on the top of DCE. They will be used both by management agents and by the application processes. The base services will be partly implemented as runtime libraries and as system processes (daemons or clerks) which will issue operating system calls and use DCE services. The base services include reliable group communication, monitoring support and basic checkpointing support.

Since DCE does not provide any form of reliable broadcast communication mechanism and it is well known that fault-tolerant programs based on replicated processes rely on such facility, we decided to implement such a service in Sampa. Although many others have also implemented reliable group communication for various kinds of operating systems and runtime environments, the main challenge in building such a service for DCE is to implement it efficiently in user space and for RPCs.

The second of the base services is the monitoring support. Depending on its purpose, monitoring can apply several techniques with different degrees of intrusion and different instrumentation requirements. However, for the purpose of availability and configuration management we found that one needs only monitoring at the process level, with simple queries to the operating system and runtime libraries for checking the availability and activity of processes and performing simple performance measurements. Such queries do not need any instrumentation of the application code and can also be implemented in user space. The basic idea is to implement the sampling, storage, filtering and pre-analysis of the monitored data in special monitoring processes, which are created by the agents, and communicate with them for setting monitoring parameters and transmitting data or events to the supervisor.

The third of the base services is a checkpointing support, which allows for saving and restoring of global variables within the application processes.[1] The basic idea is that every application process is transformed into a potential server for two basic procedures, *store_state* and *load_state*, which can be invoked either by the process itself or by any external process, e.g. an agent.

Unfortunately, checkpointing is impossible without some sort of application instrumentation. However, we chose to automate this instrumentation by a macro preprocessor for application programs written in the C language. Based on an annotation of the variables and data structures that characterize the relevant process states, this preprocessor generates the procedures to save and load the variable's values to and from a file, and also the calls to DCE's runtime library to export this interface and to wait for call requests. At runtime, procedures *load_state* and *store_state* may then be executed (by the application process itself) at user-specified program points, where the process stops and temporarily

[1] In future versions we plan to extend this checkpointing facility to record and restore also file access operations.

listens for a call request to any of the two procedures. For servers, this listening can be combined with the waiting for call requests for the other server's functions, since at these points the server is typically in a consistent state.

For all these base services, the main challenge has been to design them such that they do not require any change of DCE's runtime system and the local operating system kernel.

2.2 Agents

The management *agents* are responsible for performing all the basic and local monitoring and process control at one host on behalf of the supervisor, thus combining the functionality of Huang&Kintala's *watchdog daemon*[8] and Pilgrim's *Generic Instantiation Service*[16]. Agents are in charge of creating and destroying application processes, performing checkpoints at these processes and controlling the monitoring activities at the host. In order to perform these functions, agents use some of Sampa's base services, local operating system facilities and DCE services.

The following is a list of the three main tasks of the agents. Most of them require strong interaction with the supervisor and the other agents, which are defined in several management protocols.

Monitoring One of the main tasks of agents is to configure and control the monitoring of local processes with respect to their availability, resource usage, request and message receive rates, etc. An agent will perform monitoring control based on *monitoring commands* received from the supervisor, which may enable/disable monitoring or contain a monitoring parameter (e.g. threshold or reporting directive).

Configuration Control Agents are also responsible for creating, checkpointing, and deleting application processes, and keeping track of the current communication binding relations among these processes(by accessing DCE's *binding_reps*[17]). They perform these operations according to *configuration commands* received from the supervisor and which are derived from the availability specification for each individual program/service.

Mutual Control Because Sampa is supposed to be itself fault-tolerant, agents will periodically check for the availability of other agents and broadcast any faults to the remaining agents and the supervisor. This mutual control will be performed using a synchronous membership protocol[4] based on progressive timeouts and a cyclic organization of the agents. This mutual control protocol is described in [5].

2.3 Supervisor

The management supervisor is responsible for analyzing the monitored data and enforcing the global availability policy for each of the managed programs or services.

Since a single supervisor must be able to manage several programs within the same network, every such application will have an unique application identification (*AppId*), which will have the status of a DCE *principal*[17] and thus will be subject to the authorization and authentication procedures in DCE.

Within each application program, every process will be uniquely identified by an identification of the agent managing it (*AgentId*) and by an unique process identification (*ObjectId*). The triple (*AppId*, *AgentId* ,*ObjectId*) will thus uniquely identify every process in Sampa and will be used in every monitoring or configuration control message between the supervisor and an agent.

In some supervisor-agent interactions, some of the above mentioned identifiers may be set to a default *empty* value. For example, in an general initialization or reset request *AppId* may be empty. When *AppId* is set, but the other two fields are empty, the request is defining a global, application-specific parameter which applies to all agents. Requests with empty *ObjectId* refer to all objects of application *AppId* at agent *AgentId*.

The main tasks of a Sampa supervisor is to display the current distribution of servers and application processes in the network, accept monitoring and reconfiguration commands from the user, and control the distributed programs and services according to their specific availability specifications provided by the user. This availability specs are written in a rule-based language as event-action-pairs. In section 6 we will show a simple example of such a specification.

3 Reliable Group Communication

Reliable group communication in Sampa will be implemented using a central coordinator (sequencer) per group, similar to the approach used in Amoeba[9]. The major advantage of this approach is its simplicity and its moderate overhead compared to fully decentralized algorithms. The potential main disadvantage is that the sequencer becomes a central point of failure, requiring the election of a new sequencer and some housekeeping when the sequencer crashes.

We will overcome this problem by implementing the sequencer in a hot standby scheme. This means that it will be implemented by two processes (running on different machines) where one of them is active, i.e. is processing group communication requests, while the other continuously gets updates of the primary sequencer's state. Thus, when the primary sequencer crashes, the standby process can immediately take over, since it has all the relevant information, e.g. the message history and the acknowledgment records.

3.1 Main Characteristics

Besides the fault-tolerating characteristics mentioned above, Sampa's group communication will have following additional characteristics:

Total ordering Within each group total ordering is guaranteed by forcing that every request is handled by the corresponding sequencer, which attaches a sequence number to every broadcast or control message.

Closed and dynamic group structure In order to be able to use the group communication facilities, an application process must first join the corresponding group. Since the *join* and *leave* requests are also handled by the sequencer, this operations are synchronized with the broadcast messages.

Minimal delivery ratio When a group is created one parameter specifies the ratio of the *number of group members minus 1 (i.e the sender)* which must receive any broadcast message in the group. The ratio r determines the minimum number of group members n that must receive the messages.

FIFO delivery Given a minimal number of receivers n in a group, every message is either delivered to at least n application processes or to none of them.

Maximum Delay Another parameter of the group creation is the maximum amount of time the sender should wait for the completion of the broadcast. If the message cannot be delivered to at least n processes, the broadcast is aborted and the sender is notified.

Blocking broadcast Sending a message in a group blocks the sender until at least n application processes have received the message. Because RPC is the basic IPC mechanism in DCE, message delivery is performed as the daemon's reply to the `rcv_grp` RPC issued by the application process that is a group member.

3.2 Architecture

Unlike in Amoeba and other systems, our group communication service will be implemented through daemon processes in user mode rather than by system processes in kernel mode. Each of these daemons is a multi-threaded process which handles application processes request for any existing group. Thus, every host will have exactly one such daemon, acting as a servant (clerk) for the members of different groups.

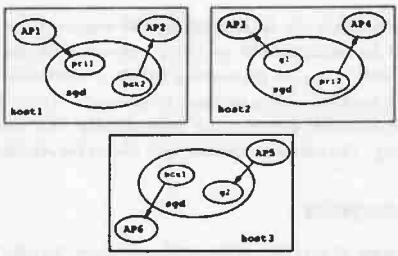

Figure 2: Sampa's group communication daemons

Figure 2 shows Sampa's group daemons (*sgd*) simultaneously serving processes in two process groups, *g1* and *g2*. In this figure, *pri*, and *bck*, denote whether the daemon plays the role of the primary or backup sequencer for a group i. While *pri* is the specific daemon at which the creation request for the corresponding group was made, *bck* is an arbitrary daemon running on another host, which is chosen at the moment the group is created.

The primary sequencer (*pri*) is in charge of processing all requests for joining or leaving, and sending group messages. These requests are made by the application processes to the local daemons, which send them to the appropriate sequencer. The *pri* daemon of a group is also responsible for storing and updating a message *history*, i.e. the messages not yet

delivered to all receivers in the group, and retransmitting a message which has not been received by a certain daemon.

Assuming n to be the minimum number of required receivers of a group, message delivery is done in a two-phase protocol, where *pri* sends a commit message to all daemons as soon as it has received acknowledgments telling that there are at least n group members waiting for the message.

There are two ways a daemon can acknowledge the receipt of a group message. In the *Direct Ack Policy* every daemon immediately acknowledges the message receipt and informs whether the message can be delivered. *Direct Ack* is automatically set if the group has been created with a non-zero delay parameter. The second form is the *Indirect Ack Policy*, in which the daemon waits until there is another message to *prim*, on which it can piggy-back the acknowledgment.

In both cases, the acknowledgment consists of a pair of sequence numbers, one for the last received (`rcv_ack`) and for the last delivered message(`dlv_ack`), and an integer `dlv_nr` specifying how many group members are waiting for a broadcast at the daemon's site. With `rcv_ack` *pri* knows if it has to retransmit a given message to a certain daemon. Sequence number `dlv_ack` and the sum of `dlv_nr` from all daemons tell *pri* if the message can be delivered to at least n group members, and thus if it can unblock the sender of the group message.

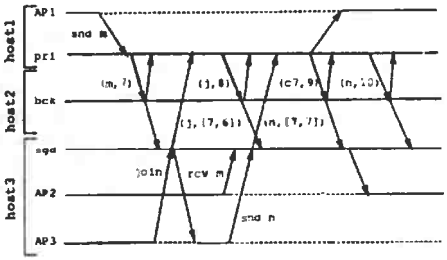

Figure 3: Group communication with Indirect Ack Policy

Figure 3 shows the control messages among the daemons (`pri`, `bck`, `sgd`) for a single group (with members `AP1`, `AP2` and later `AP3`) using the *Indirect Ack Policy*. In this example `AP1` sends message `m`, which is later received by `AP2`. Every message from `pri` carries a sequence number, and messages from other daemons to *pri* carry the pair of sequence numbers `rcv_ack` and `dlv_ack`, respectively, shown in square braces.

Notice that since the `rcv_grp` has been received by `sgd` after the join request from `AP3`, the first piggy-backed `dlv_ack` is still 6, while `rcv_ack` is 7. With this information `pri` knows that it need not retransmit message `m` to `sgd`, but it cannot release `AP1` until it receives `dlv_ack` with value 7. Only when `pri` gets the ack-pair `[7,7]`, it knows that there is a receiver at `sgd`. It then can unblock `AP1` and also send the commit for message `m` (`c7`), by which `sgd` can then deliver the message to `AP2`. Notice also that since `AP3` joined the group after the broadcast of `m`, it should not receive the message.

As suggested by the previous example, the backup daemon *bck* is required to use always

the *Direct Ack Policy* to acknowledge *pri*'s state update, and if *bck* also has any group members its `dlv_ack` and `dlv_nr` will also be taken into account by *pri*. Although this hot standby scheme represents an additional overhead for the group service, it pays off when *pri* crashes, because then *bck* can immediately assume the primary role informing all other daemons of this change, choosing a new backup and sending the current message history to this new backup.

3.3 Interface

The group communication service of Sampa is described by following RPC interface, which is to be imported by every application process enrolling in a group membership.

```
create_grp([in] mx_mem, [in] mx_msgs, [in] ratio, [in] mx_delay) — gr_id
join_grp ([in] gr_id, [in] Pid):— bool
leave_grp([in] gr_id, [in] Pid):— bool
snd_grp ([in] gr_id, [in] msg, [in] msg_size):— bool
rcv_grp ([in] gr_id, [out] buffer, [in] buff_size):— int
info_grp([in] gr_id, [in] kind, [out] buffer, [in] buff_size):— bool
reset_grp([in] gr_id, [in] ratio, [in] mx_mem, [in] mx_delay):— bool
```

Besides the already mentioned parameters *minimal delivery ratio* (ratio) and *maximum delay* (mx_delay), procedure create_grp is also parameterized with the maximum number of group members (mx_mem) and the size of the message history (mx_msgs). The return value is a unique group identifier, which must be used in all other calls to the group communication service. Notice that it is the task of the application program to disseminate this group identifier among all processes that are candidates for becoming group members. Procedure info_grp is used to obtain different *kinds* of information about a group, e.g. the identification of its *pri* (or *bck*) daemons, the list of member processes, the current size of its message history, the number of aborted broadcasts, etc.

Besides this external interface, the group communication service has also an internal interface, which defines the calls among the daemons for control purposes. However, since in this case it is necessary to address individual daemons, there will be a specific CDS entry for each of the daemons.

4 Monitoring Support

Since monitoring in Sampa is focused on availability management, it requires only facilities for collecting process-level information i.e. data and events that are visible externally to the processes and can be collected asynchronously to the processes execution. Examples of such data are CPU utilization, process activity or RPC request rates, etc. Thus, Sampa's monitoring facilities will be restricted to the sampling and analysis of data that can be obtained by queries to the local operating system and the DCE runtime library. Such *external monitoring* has the advantage of being less intrusive than other approaches which instrument application processes and runtime libraries with probes that send runtime data to sensors. The other advantage of *external monitoring* is that it does not require any

changes to the operating system or runtime libraries, which makes it more portable than other techniques.

4.1 Basic Concepts

We define *monitoring instances* to be active entities at one host which collect runtime data either related to the host or to an individual application process running on that host.

Every monitoring instance is derived from a *monitoring type*, which defines a monitoring metric, i.e. the type of data collected and the parameters that can be set for every instance of this type, such as the sampling interval, a threshold, or the precision.

Examples of monitoring types could be programs that query the operating system, like `iostat`, `vmstat`, the DCE runtime library and DCE services, or poll the activity of an application process by sending it signals[2]. Every *type* provides a specific kind of monitored data, which can range from a simple integer to a structured set of data.

4.2 Architecture

As mentioned, all monitoring activities at a host will be controlled by the corresponding agent on that host, which will create monitoring instances each of which will be responsible for monitoring a particular instance of a host, operating system or DCE resource, according to the functionality defined by their corresponding type. Instances will sample performance and resource utilization data by making the appropriate calls to the operating system and shared DCE runtime libraries, store the data in internal data structures and communicate with the controlling agent whenever data or an event needs to be forwarded to the supervisor. Communication between the agent and its monitoring instances will be performed using the local IPC mechanism, such as the *UNIX pipe*.

Figure 4 illustrates this monitoring architecture, where *I1*, *I2* and *I3* are monitoring instances created for different metrics.

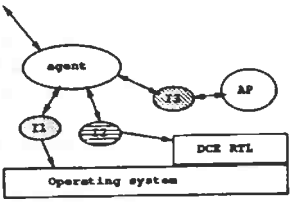

Figure 4: Sampa's Monitoring Architecture

All local monitoring control done by the agents will be done according to *monitoring commands* received from the supervisor. Through these monitoring commands the supervisor may enable or disable monitoring of a particular instance, set threshold values and sampling intervals, or set other monitoring parameters. Most monitoring commands will include a unique identification of the monitoring instance (*inst*), which will have been assigned at

[2] In this case however, the program must be instrumented to react to the signals

creation time using the naming convention discussed in section 2.3. Some commands may also contain the identification of the object *obj*(i.e. process) to be monitored. At instance creation, *mtype* indicates the executable file that implements the corresponding monitoring type.

Following is a list of some monitoring commands and their semantics:

crt *mtype* → *inst*	start *mtype*
ena *inst* [*obj*]	enable monitoring of *obj* by *inst*
dis *inst* [*obj*]	disable monitoring of *obj* by *inst*
set_thr *inst* [*obj*] *value*	set a threshold
set_si *inst* [*obj*] *time*	set the sampling interval (in sec)
set_ni *inst* [*obj*] *time*	set time interval for notifications to supervisor (in sec)

Recall that depending on the type, only some of the monitoring commands for setting parameters may are applicable, and the others will be ignored by the agent.

The main advantage of this monitoring architecture is that since *monitoring instances* are actually processes *fork*-ed and *exec*-ed by the agent, this yields to a flexible monitoring approach where new, and operating system-specific *types* can be easily incorporated. The other advantage is the uniform treatment among monitoring and application processes within Sampa.

The obvious disadvantage is that having additional processes for monitoring, their resource usage will have a significant influence on the system's performance. The other disadvantage is the lack of control on the scheduling of the monitoring instances (done by the operating system), which makes it impossible to enforce precise sampling and notification intervals.

5 Checkpointing Support

Sampa's checkpointing support is based on the premise that state-saving operations are essentially application-specific and should therefore incorporate much of the application programmer's knowledge of which are the data structures within each process that fully characterize the program state.

Therefore we decided to take a simple approach of implementing a source-code preprocessor which automatically generates procedures *save_state* and *load_state*, that store and load the values of annotated variables to and from a file[3] provided as a parameter to these procedures. Since these procedures are also exported to the CDS namespace, they can be invoked both by the checkpointing process itself and by other processes, as for example, the agents. Although the above procedures have exactly the same signature for all processes, are defined in an unique interface definition file `spcp.idl` and thus share the same *UUID*, their definition will differ from one process to another due to the specific data structures to be saved in each process.

For another process to be able to identify these procedures at different processes, the corresponding bindings are exported as different entries to the CDS with Sampa's unique process identification as a namespace entry attribute. When a process needs to save or

[3]Hence our approach requires the existence of a distributed file system like DCE's DFS

reload its own state, however, it should call its own procedures rather than the equivalent remote procedures.

5.1 The preprocessor

In this section we present some of the preprocessor macros and explain how the preprocessor works along a simple example.

As mentioned earlier, the source code of an application process must be slightly modified in order to allow for its state to be checkpointed. These changes include the annotation of some variables and data structures (with a tilde ~) as well as macro calls at the program points where checkpointing should be initialized (*INITCP*) and where checkpointing is enabled (*CPHERE*).

Figure 5 shows a simple example of an annotated source code. In this case, variables `state` and `list` represent the process state to be checkpointed.

```
main(int argc, char **argv)
{
  int ~state;
  reg_t* ~list;
  INITCP
  while (1) {
    CPHERE
    /* other processing */
  }
}
```

Figure 5: Annotated code

After running the checkpoint preprocessor over this source, the original source code is expanded as shown in Figure 6. The main extensions are the definition and call of functions to initialize and wait for checkpointing requests to procedures *save_state* and *load_state*, which have also been generated. Besides this, variables `state` and `list` have been transformed into global variables in order to be accessible by these procedures and the checkpoint-specific (and *idl*-generated) `spcp.h` file has been included. Notice that *CPHERE* has been substituted by a call to DCE's blocking `rpc_server_listen(...)`[4] and the starting of a thread which will unblock our program after time (`T`) by calling DCE's runtime procedure `rpc_mgmt_stop_server_listening`. Notice that `T` must be chosen big enough to allow for both `save_state` or `load_state` to complete. In the generated procedure `save_state`, `sv_int` is a library procedure for saving integer values (in a canonic format) to the file and `sv_lregt` is a generated procedure for saving a linked data structure based on `rec_t`, which uses library procedures for the scalar types within this structured type.

Sampa's checkpointing approach thus requires the application programmer to decide only which data structures characterize the relevant process state, and to choose the program point(s) where checkpointing should be allowed. For server processes that also export

[4] If local checkpointing should also be done at this point, `save_state` must be called after this command.

```
#include "spcp.h"
int state;
reg_t* list;
void sp_cp_init(void);
void listen_until(int t);
void save_state(char *filename);
void load_state(char *filename);

main(int argc, char *argv[])
{
  sp_cp_init;
  while (1)
  {
    pthread_create(listen_until(T));
    rpc_server_listen(..)
   /*  other processing */
  }
}
void save_state(char *file) {
  FILE *fd;
  fd = fopen(file,"w");
  sv_int(fd, state);
  sv_lregt(fd, list);
  fclose(fd);
}
..definition of the other procedures ..
```

Figure 6: Expanded Source Code

another interface, the programmer even must not set CPHERE, since the process can listen to all its interfaces (including the checkpoint interface) simultaneously.

6 Availability Specification

In this section we give an idea of Sampa's language for specifying the availability policy of distributed programs, along a simple example.

It is a rule-based scripting language with Perl[18]-like control structures, a set of basic operators for comparison and manipulation of strings, list and sets, such as $<$, $==$ (equality), $++$ and $--$ (set union and difference), and others. Besides those, there is a set of build-in primitives for performing basic configuration and monitoring control, which are translated into equivalent supervisor commands to the agents.

The following table gives the semantics of some common primitives for different purposes, where *prog*, *proc*, *grp* and *itf* are place holders for the name of an executable (or its call), a process, a group and an interface identifier, respectively, and the suffix L always denotes list of such names/identifiers.

Configuration control:	
Create(prog,host) —proc	create new process from *prog* at *host*
Delete(proc)	delete a process
GetProcId(prog,host) —procL	all processes of *prog* executing at *host*
GetGroupId(proc) —grp	groupId created by *proc*
GetBindings(procL,itfL) —procL	all processes with bindings to any interface in *itfL* of *procL*
Monitoring Control:	
Crashed(hostL) —host	event of a crashed host among *hostL*
Lost(prog,hostL) —procL	event of process crash on any of *hostL*
Stopped(prog,hostL) —procL	event of process hang-up on any of *hostL*
Miscellaneous:	
Any(list) —elem	selects any element from *list*
Card(list) —integer	cardinality of *list*

Every rule in this script language is an event-action-pair written in the form *event >> actions*. An event may be the occurrence of a failure, the satisfaction of a condition involving a boolean expression and script variables, or the passing of a given time period. The latest case is used for specifying periodic control actions. There is also `init`, which is a special event that is triggered by an interactive user command to start a distributed program.

The language also supports script variables (preceded by %), whose values are list of names, identifiers or integers, all of them represented as strings. As in most scripting languages, those variables don't have to be declared before they are used. Some special (environment) variables have predefined values which are set when Sampa is started, such as `%AllHosts`, which contains the list of all managed hosts. However, most of these variables can be redefined for each application.

6.1 Example

In this section we will give an example of an availability specification for a hypothetic fault-tolerant service implemented using the active replica approach[13]. In this approach, a service is implemented by replicated servers, which maintain their states exactly the same and synchronized with the receipt of any new request. As mentioned, active replica is usually implemented by connecting the servers through a communication group with the property of FIFO delivery.

Assume that for our replicated service (`ARserv`) we want to detect if some of the servers has crashed, and if this happened, to start a new server which should incorporate the other replica's state and also join the communication group. For this to be possible, we assume that at least one of the servers checkpoints its relevant state to a given file (with its name stored in script variable `%ARfile`) after processing each of its requests. Thus, assuming that the group has been created with *ratio = 1* delivery of every broadcast message is postponed until the minimum number of members is available. Therefore, if a new member starts with the current state of the service, i.e. the checkpointed state, and the other members can proceed with servicing requests only when the new member joins the group, we guarantee that the group of replicated servers will also have synchronized states in this new 'incarnation' of the service.

```
ARserv {
init >> {
       %GrCreator = Create("ARserver -g","mafalda");
       Create("ARserver","bidu");
       Create("ARserver",Any(%AllHosts));
       %gid = GetGroupId(GrCreator);
      }
+10 >> {
       %gr_memb = GetGrpInf(%gid,"all_members");
       %servers = "";
       forall %j in %AllHosts do
            %servers = %servers ++ GetProcId(ARserver,%j);
      }
Card(%servers) < Card(%gr_memb) >> {
       %missing = %gr_memb -- %servers;
       forall %i in %missing do {
            %new = create("ARserver",Any(%Allhosts));
            load(%new,%ARfile);
            join_grp(%gid,%new);
            leave_grp(%gid,%i);
        }
      }
```

Figure 7: Sampa Script for Replicated Servers

Figure 7 shows the script describing the monitoring and control actions that gives one solution for enforcing the above-mentioned availability policy for service `ARserv`. At event `init` the service is started with three servers, of which the first server is the group creator. The group id is obtained from this process and stored in variable `%gid`.

The remaining two sections of the script specify that every 10 seconds the system must check if the number of servers (instances of program `ARserver`) is equal to the number of group members. The set of group members is obtained form `GetGrpInfo`, which is translated into a command from the supervisor to an agent for calling `info_grp`. Primitive `GetProcId` is used to get the process Ids of all executing servers on each node, which are then collected in variable `%servers`.

Once number of replicas is less than the number of group members (`Card(%servers) < Card(%gr_memb)`), the set difference is calculated and stored in list `%missing`. Then, for each list element, a new servers is created, which is initialized with the checkpointed state and joined to the communication group, and the list element is unregistered from the group.

Although this language has not yet been completely defined, the example should give an idea of how it can be used to describe reconfigurations that ensure continuous availability for some class of fault-tolerant distributed programs.

7 Related Work

Several other groups working with monitoring, fault-tolerance, dynamic configuration and distributed application management have had influence on the design of Sampa's architecture and services.

CONIC[11] and its successor languages [10] have shown the advantages of having a strict separation between the program algorithms and configuration. With Sampa, we aim at extending this approach for availability policies for fault-tolerant programs. Bauer et al. proposed a reference architecture for general-purpose distributed application management[1], but their main focus is on monitoring, management integration and system modeling.

Becker[2] describes an architecture based on the concept of a *fault-tolerance layer* that hides fault-tolerance issues from distributed programs. Similar to Sampa's base services and agents, this layer provides special services (e.g. surveillance, checkpointing, atomic broadcast) for services with fault-tolerance requirements. The main difference is that in his approach, the availability policy is programmed into such layer, rather than being input for a supervising program.

Specific tools for monitoring and controlling distributed applications have also been implemented. One of the them is the Megascope[15] of the Pilgrim Project[14]. It is a basic monitoring service for management of DCE-based applications, which is designed around a centralized database (panel), for collecting and querying cell-specific monitored data.

Huang&Kintala[8] have also implemented a nice set of tools for controlling availability and checkpointing, but their work is not concerned with describing global availability specifications.

Another one the Meta Toolkit[12], which is based on the ISIS system[3]. It provides means for instrumenting distributed application processes with *sensors* and *actuators*, which are used for monitoring and controlling the execution of the application processes from a control layer, where monitoring & configuration is specified in a rule-based language (Lomita). Compared to Meta, our work differs in that it allows for a less intrusive and more flexible monitoring and defines a higher-level language for specifying availability and monitoring directives.

8 Conclusion

The Sampa project started from the belief that distributed applications, in particular those with high availability requirements, require tools which allow for automating (at least some of) its reconfiguration and recovery actions. Since OSF's DCE is emerging as a de-facto standard environment for developing distributed programs, we decided to design such a system for DCE-based programs/services.

We have just started implementing the base services, in particular the group communication and monitoring services. For both services, we noticed that implementing them only with DCE's RPC would lead to poor performance. Therefore, we decided to implement these services using Concert/C[6], which provides both message passing and RPC facilities, and gives means for interfacing DCE-based applications. Our decision to implement monitoring as a flexible service performed by processes came from the understanding that monitoring is always very dependent on the particular execution environment, and that one should be

able to adapt monitoring to the specific operating systems in use. Unfortunately, we have not yet started the implementation of Sampa's checkpointing facility, but we believe that this will happen in the next months.

In parallel to those developments, we are designing the supervisor-agent protocols for monitoring and configuration control and implementing some agent prototypes. By the middle of next year we expect to have an agent implementation with a reasonable set of monitoring and configuration control facilities and then we will finish the definition of Sampa's scripting language and implement the supervisor.

We have not yet tackled many other important issues, such as security management, support for multiple management, and scalability issues, which we will consider in later steps of the project.

References

[1] M.A. Bauer, P.F. Finnigan, J.W. Hong, J.A. Rolia, T.J. Teorey, and G.A. Winters. Reference Architecture for Distributed Systems Management. *IBM Systems Journal*, 33(3):426–444, 1994.

[2] T. Becker. Application-Transparent Fault Tolerance in Distributed Systems. In *Proc. of 2nd. Int. Workshop on Configurable Distributed Systems*, pages 36–45, March 1994.

[3] K.P. Birman and R. Cooper. The ISIS Project: Real Experience with a Fault Tolerant Programming System. *Operating Systems Review*, 25(2):103–107, April 1990.

[4] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):57–78, February 1991.

[5] M. Endler. The Design of Sampa. In *Proc. 2nd International Workshop on Services in Distributed and Networked Environments, Whistler, CA*, pages 86–92. IEEE, June 1995.

[6] J.S. Auerbach et all. Concert/C Tutorial and User Guide: An Introduction to a Language for Distributed C Programming. Technical report, IBM T. J. Watson Research Center, January 1995.

[7] J.W. Hong and M.A. Bauer. A Generic Management Framework for Distributed Applications. In *Proc. 1st Int. Workshop on System Management*, pages 63–71. IEEE, April 1993.

[8] Y. Huang and C. Kintala, *Software Fault Tolerance in the Application Layer*, chapter 10. John Wiley & Sons, 1995.

[9] M.F. Kaashoek, A.S. Tanenbaum, and K. Verstoep. Group Communication in Amoeba and its Application. *Distributed Systems Engineering Journal*, 1(1):48–58, July 1993.

[10] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. In *Proc. of the Int. Workshop on Configurable Distributed Systems*, pages 102–117. IEE, March 1992.

[11] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, SE-15(6), June 1989.

[12] K. Marzullo, R. Cooper, M.D. Wood, and K.P. Birman. Tools for Distributed Application Management. *IEEE Computer*, 24(8):42–51, August 1991.

[13] S. Mullender. *Distributed Systems*. Addison Wesley, 1993.

[14] J.D. Narkiewicz, M. Girkar, M. Srivastava, A.S. Gaylord, and M. Rahman. Pilgrim OSF DCE-based Services Architecture. In *Proc. of International DCE Workshop, LNCS 731*, pages 120–134, October 1993.

[15] B. Obrenić, K.S. DiBella, and A.S. Gaylord. DCE Cells under Megascope: Pilgrim Insight into the Resource Status. In *Proc. of International DCE Workshop, LNCS 731*, pages 162–178, October 1993.

[16] L.J. Padmanaban, K.S. DiBella, and A.S. Gaylord. A Generic Instantiation Service. Tech. report, University of Massachusetts at Amherst, 1995. http://www.pilgrim.umass.edu/pub/pilgrim/papers/gis/gis.

[17] J. Shirley, W. Hu, and D. Magin. *Guide to Writing DCE Applications*. O'Reilly & Associates Inc., 1994.

[18] L. Wall and R.L. Schwarz. *Programming Perl*. O'Reilly & Associates Inc., 1991.

RELATÓRIOS TÉCNICOS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Instituto de Matemática e Estatística da USP

A listagem contendo os relatórios técnicos anteriores a 1992 poderá ser consultada ou solicitada à Secretaria do Departamento, pessoalmente, por carta ou e-mail(mac@ime.usp.br).

J.Z. Gonçalves, Arnaldo Mandel
COMMUTATIVITY THEOREMS FOR DIVISION RINGS AND DOMAINS
RT-MAC-9201, Janeiro 1992, 12 pp.

J. Sakarovitch
THE "LAST" DECISION PROBLEM FOR RATIONAL TRACE LANGUAGES
RT-MAC 9202, Abril 1992, 20 pp.

Valdemar W. Setzer, Fábio Henrique Carvalheiro
ALGOTRITMOS E SUA ANÁLISE (UMA INTRODUÇÃO DIDÁTICA)
RT-MAC 9203, Agosto 1992, 19 pp.

Claudio Santos Pinhanez
UM SIMULADOR DE SUBSUMPTION ARCHITECTURES
RT-MAC-9204, Outubro 1992, 18 pp.

Julio M. Stern
REGIONALIZAÇÃO DA MATRIZ PARA O ESTADO DE SÃO PAULO
RT-MAC-9205, Julho 1992, 14 pp.

Imre Simon
THE PRODUCT OF RATIONAL LANGUAGES
RT-MAC-9301, Maio 1993, 18 pp.

Flávio Soares C. da Silva
AUTOMATED REASONING WITH UNCERTAINTIES
RT-MAC-9302, Maio 1993, 25 pp.

Flávio Soares C. da Silva
ON PROOF-AND MODEL-BASED TECHNIQUES FOR REASONING WITH UNCERTAINTY
RT-MAC-9303, Maio 1993, 11 pp.

Carlos Humes Jr., Leônidas de O.Brandão, Manuel Pera Garcia
A MIXED DYNAMICS APPROACH FOR LINEAR CORRIDOR POLICIES
(A REVISITATION OF DYNAMIC SETUP SCHEDULING AND FLOW CONTROL IN MANUFACTURING SYSTEMS)
RT-MAC-9304, Junho 1993, 25 pp.

Ana Flora P.C.Humes e Carlos Humes Jr.
STABILITY OF CLEARING OPEN LOOP POLICIES IN MANUFACTURING SYSTEMS (Revised Version)
RT-MAC-9305, Julho 1993, 31 pp.

Maria Angela M.C. Gurgel e Yoshiko Wakabayashi
THE COMPLETE PRE-ORDER POLYTOPE: FACETS AND SEPARATION PROBLEM
RT-MAC-9306, Julho 1993, 29 pp.

Tito Homem de Mello e Carlos Humes Jr.
SOME STABILITY CONDITIONS FOR FLEXIBLE MANUFACTURING SYSTEMS WITH NO SET-UP TIMES
RT-MAC-9307, Julho de 1993, 26 pp.

Carlos Humes Jr. e Tito Homem de Mello
A NECESSARY AND SUFFICIENT CONDITION FOR THE EXISTENCE OF ANALYTIC CENTERS IN PATH FOLLOWING METHODS FOR LINEAR PROGRAMMING
RT-MAC-9308, Agosto de 1993

Flavio S. Corrêa da Silva
AN ALGEBRAIC VIEW OF COMBINATION RULES
RT-MAC-9401, Janeiro de 1994, 10 pp.

Flavio S. Corrêa da Silva e Junior Barrera
AUTOMATING THE GENERATION OF PROCEDURES TO ANALYSE BINARY IMAGES
RT-MAC-9402, Janeiro de 1994, 13 pp.

Junior Barrera, Gerald Jean Francis Banon e Roberto de Alencar Lotufo
A MATHEMATICAL MORPHOLOGY TOOLBOX FOR THE KHOROS SYSTEM
RT-MAC-9403, Janeiro de 1994, 28 pp.

Flavio S. Corrêa da Silva
ON THE RELATIONS BETWEEN INCIDENCE CALCULUS AND FAGIN-HALPERN STRUCTURES
RT-MAC-9404, abril de 1994, 11 pp.

Junior Barrera; Flávio Soares Corrêa da Silva e Gerald Jean Francis Banon
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES
RT-MAC-9405, abril de 1994, 15 pp.

Valdemar W. Setzer; Cristina G. Fernandes; Wania Gomes Pedrosa e Flavio Hirata
UM GERADOR DE ANALISADORES SINTÁTICOS PARA GRAFOS SINTÁTICOS SIMPLES
RT-MAC-9406, abril de 1994, 16 pp.

Siang W. Song
TOWARDS A SIMPLE CONSTRUCTION METHOD FOR HAMILTONIAN DECOMPOSITION OF THE HYPERCUBE
RT-MAC-9407, maio de 1994, 13 pp.

Julio M. Stern
MODELOS MATEMATICOS PARA FORMAÇÃO DE PORTFÓLIOS
RT-MAC-9408, maio de 1994, 50 pp.

Imre Simon
STRING MATCHING ALGORITHMS AND AUTOMATA
RT-MAC-9409, maio de 1994, 14 pp.

Valdemar W. Setzer e Andrea Zisman
*CONCURRENCY CONTROL FOR ACCESSING AND COMPACTING B-TREES**
RT-MAC-9410, junho de 1994, 21 pp.

Renata Wassermann e Flávio S. Corrêa da Silva
TOWARDS EFFICIENT MODELLING OF DISTRIBUTED KNOWLEDGE USING EQUATIONAL AND ORDER-SORTED LOGIC
RT-MAC-9411, junho de 1994, 15 pp.

Jair M. Abe, Flávio S. Corrêa da Silva e Marcio Rillo
PARACONSISTENT LOGICS IN ARTIFICIAL INTELLIGENCE AND ROBOTICS.
RT-MAC-9412, junho de 1994, 14 pp.

Flávio S. Corrêa da Silva, Daniela V. Carbogim
A SYSTEM FOR REASONING WITH FUZZY PREDICATES
RT-MAC-9413, junho de 1994, 22 pp.

Flávio S. Corrêa da Silva, Jair M. Abe, Marcio Rillo
MODELING PARACONSISTENT KNOWLEDGE IN DISTRIBUTED SYSTEMS
RT-MAC-9414, julho de 1994, 12 pp.

Nami Kobayashi
THE CLOSURE UNDER DIVISION AND A CHARACTERIZATION OF THE RECOGNIZABLE Z-SUBSETS
RT-MAC-9415, julho de 1994, 29pp.

Flávio K. Miyazawa e Yoshiko Wakabayashi
AN ALGORITHM FOR THE THREE-DIMENSIONAL PACKING PROBLEM WITH ASYMPTOTIC PERFORMANCE ANALYSIS
RT-MAC-9416, novembro de 1994, 30 pp.

Thomaz I. Seidman e Carlos Humes Jr.
SOME KANBAN-CONTROLLED MANUFACTURING SYSTEMS: A FIRST STABILITY ANALYSIS
RT-MAC-9501, janeiro de 1995, 19 pp.

C.Humes Jr. and A.F.P.C. Humes
STABILIZATION IN FMS BY QUASI- PERIODIC POLICIES
RT-MAC-9502, março de 1995, 31 pp.

Fabio Kon e Arnaldo Mandel
SODA: A LEASE-BASED CONSISTENT DISTRIBUTED FILE SYSTEM
RT-MAC-9503, março de 1995, 18 pp.

Junior Barrera, Nina Sumiko Tomita, Flávio Soares C. Silva, Routo Terada
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES BY PAC LEARNING
RT-MAC-9504, abril de 1995,16 pp.

Flávio S. Corrêa da Silva e Fabio Kon
CATEGORIAL GRAMMAR AND HARMONIC ANALYSIS
RT-MAC-9505, junho de 1995,17 pp.

Henrique Mongelli e Routo Terada
ALGORITMOS PARALELOS PARA SOLUÇÃO DE SISTEMAS LINEARES
RT-MAC-9506, junho de 1995, 158 pp.

Kunio Okuda
PARALELIZAÇÃO DE LAÇOS UNIFORMES POR REDUCÃO DE DEPENDÊNCIA
RT-MAC-9507, julho de 1995, 27 pp.

Valdemar W. Setzer e Lowell Monke
COMPUTERS IN EDUCATION: WHY, WHEN, HOW
RT-MAC-9508, julho de 1995, 21 pp.

Flávio S. Corrêa da Silva
REASONING WITH LOCAL AND GLOBAL INCONSISTENCIES
RT-MAC-9509, julho de 1995, 16 pp.

Julio M. Stern
MODELOS MATEMÁTICOS PARA FORMAÇÃO DE PORTFÓLIOS
RT-MAC-9510, julho de 1995, 43 pp.

Fernando Iazzetta e Fabio Kon
A DETAILED DESCRIPTION OF MAXANNEALING
RT-MAC-9511, agosto de 1995, 22 pp.

Flávio Keidi Miyazawa e Yoshiko Wakabayashi
POLYNOMIAL APPROXIMATION ALGORITHMS FOR THE ORTHOGONAL Z-ORIENTED 3-D PACKING PROBLEM
RT-MAC-9512, agosto de 1995, pp.

Junior Barrera e Guillermo Pablo Salas
SET OPERATIONS ON COLLECTIONS OF CLOSED INTERVALS AND THEIR APPLICATIONS TO THE AUTOMATIC PROGRAMMINIG OF MORPHOLOGICAL MACHINES
RT-MAC-9513, agosto de 1995, 84 pp.

Marco Dimas Gubitoso e Jörg Cordsen
PERFORMANCE CONSIDERATIONS IN VOTE FOR PEACE
RT-MAC-9514, novembro de 1995, 18pp.

Carlos Eduardo Ferreira e Yoshiko Wakabayashi
ANAIS DA I OFICINA NACIONAL EM PROBLEMAS COMBINATÓRIOS: TEORIA, ALGORITMOS E APLICAÇÕES
RT-MAC-9515, novembro de 1995, 45 pp.

Markus Endler and Anil D'Souza
SUPPORTING DISTRIBUTED APPLICATION MANAGEMENT IN SAMPA
RT-MAC-9516, novembro de 1995, 22 pp.