21

A high-level process checkpointing and migration scheme for heterogeneous distributed systems

Tim Redhead
University of Queensland, Australia
CRC for Distributed Systems Technology, Level 7, Gehrmann Labs,
University of Queensland, Australia, 4072,
email: `redhead@dstc.edu.au`

Abstract
Reliability is a key concern of designers of distributed computing systems. Checkpointing can be used as a basis for designing resilient processes and process-migration schemes, but very few systems which implement process-checkpointing are heterogeneous. 'High-level' process checkpointing schemes capture process-state at a higher level of abstraction than do low-level schemes. The resulting state does not depend on low-level or platform-specific structures, and so is meaningful at any site in a heterogeneous distributed computing network. This paper presents a high-level approach to process checkpointing which is transparent to the programmer, which operates at a fine level of granularity, and which can deal with dynamically allocated memory and multithreaded processes.

Keywords
Process, checkpoint, heterogeneous, resilient, distributed.

1 INTRODUCTION

Process *checkpointing* involves capturing the state of a process in a single, atomic action. *Restoration* is the inverse operation of checkpointing and involves returning the captured state to the process, which can then continue executing as it would have if the checkpoint had never taken place. Checkpointing is often used as a basis for both process migration and the implementation of *resilient processes* (Mishra and Sclichting, 1992).

A heterogeneous distributed computing system is one where each of the machines in the system may have different hardware architectures *and* operating systems. Many existing systems which provide process checkpointing and migration mechanisms are described by Nuttall (1992). These and other systems can be broadly categorised according to the degree of heterogeneity of the environment for which they are designed. The first group includes distributed operating systems such as Amoeba (van Renesse, van Staveren and Tanenbaum, 1988), and operates in a totally homogeneous environment. It also includes distributed systems such as Argus (Bal, 1992), which require both the operating system and the hardware on all machines be identical (Argus is implemented on a system of VAX3200's, which all run the UNIX operating system). The second group of systems include those that are designed for a partially heterogeneous environment; either the hardware platforms or the operating systems at each site may be different, but not both.

1.1 The low-level approach

None of the systems that are members of the first two groups are useful in a heterogeneous distributed environment. In contrast, the third group consists of those systems that are designed to run in a totally heterogeneous environment, and includes systems such as Chameleon (Attardi. et. al, 1987(a), 1987(b), 1988). However, Chameleon is also an example of a system which is based on a *low-level* checkpointing approach. The low-level approach suffers from the following shortcomings, when used as a basis for resilient processes and process migration in a heterogeneous system: machine-level structures such as stacks and registers must be translated or interpreted, operating systems can be widely dissimilar or incompatible, and the semantics of any state-data must be maintained so that the process-state is consistent at any host in the distributed system.

1.2 The high-level approach

The work presented in this paper aims to develop an alternative to the low-level approach to process checkpointing. *High-level* process checkpointing does not rely on any platform-specific variables or structures. Instead, process-state is captured at the language-level of abstraction. The captured state is meaningful at any site in a heterogeneous distributed system, since it exists in a platform-independent format.

The checkpointing and migration system presented in this paper is based on a high-level process-checkpointing mechanism and is designed to function in a heterogeneous, distributed environment. This system, known as HiCaM (for High-level Checkpointing and Migration), avoids the difficulties and inefficiencies of translating or interpreting a low-level process-state as it is moved between dissimilar sites.

Other schemes which have taken a similar, high-level approach include Arjuna (Shrivastava and Parrington, 1991), DC++ (Schill and Mock, 1993) and ANSAware (ANSAware, 1993), but the aim of this work is to improve on these schemes in a number of areas. Specifically, the aims of HiCaM are to:

- Reduce the additional work required of programmers and designers.
- Reduce the risk of errors in the definition of the process-state
- Reduce the level of granularity of the checkpoint operation.
- Allow the high-level checkpointing of dynamically allocated memory.
- Enable the high-level checkpointing of multithreaded processes.

1.3 Outline of this paper

This paper describes the design of a system which provides the infrastructure for process migration and resilient processes in a heterogeneous distributed system. Section two of this paper discusses several design principles and gives an overall view of the whole high-level scheme, describing the general environment and how each of the components of the system work together. Section three describes OSF DCE. Sections four to eight present the design and current implementation of each of the system components in more detail. Individual system components include preprocessor tools as well as runtime support applications. Section nine outlines future work and section ten summarizes the work presented in this paper and offers some conclusions about what has been achieved to date. An implementation overview is given at the end of each relevant section, but the reader is referred to (Redhead, 1995) for more detailed implementation information.

2 PRINCIPLES OF THE SYSTEM DESIGN

This section provides an overview of the design of the high-level checkpointing system. There are three main principles behind the design presented here.

The first principle of this work is that process-state should be captured at a level which is not reliant on platform-specific factors. Platform-specific factors include the number of registers present in the hardware of a particular machine, or some intrinsic feature of a particular operating system, such as its ability to dump core in response to a software-interrupt. Such platform-specific features would severely limit a system's usefulness in a heterogeneous environment, since there is no guarantee that the feature will be present at all the system sites.

Application processes are viewed as high-level abstract machines. This view-point excludes platform-specific characteristics from the definition of an abstract machine's state. However, this approach assumes that an abstract machine exists in a suitable distributed computing environment, as described in section 3. There are a number of environments which support these features, including OSF's DCE (Shirley, 1992), (Rosenberry, 1992), which has been used to support the HiCaM system presented here.

As described by Theimer and Hayes (1991), a high-level programming language defines an abstract machine. Compilers are used to translate the platform-independent source (abstract-machine) code to platform dependant binary (physical-machine) code, but the behaviour (as described below) which is described by both source object and binary code is the same. In addition, there will be points in the program's execution, termed migration-points, where the process state can be specified in terms of the current state of the abstract machine. The more frequent the migration points, the less the delay in waiting for a migration call to be completed.

The second principle behind the work presented here, is that checkpoint/restore functionality must be included as part of an application process, rather than as part of an operating system. The individual functionality that is provided by the HiCaM mechanism is different for each application; the functions which will checkpoint and restore a process are based on the process's state, and that state is defined in terms of the high-level programming language. Since the functionality is application-specific, it makes sense to include the functionality within the application itself. In addition, processes cannot rely on underlying operating system-specific mechanisms in a heterogeneous distributed network, since that feature will almost certainly be absent at some of the system sites.

The third principle of this work is that the system is designed from an object perspective. Processes are considered to be objects, each with hidden mechanism and at least one, well-defined interface. All inter-object communication (including file and database I/O) must take place through the object's interface(s), via a remote-procedure call mechanism. Object-behaviour is described by the state of the object which is visible at the object's interface. In this design, the object-paradigm is extended to apply at compile-time as well as at run-time, so that application-designers and programmers need not be aware of the mechanisms which will eventually allow their objects to be checkpointed or migrated; pre-compiler tools automatically add any checkpoint/ restore functionality (along with other functions which are described below) to the general, application code.

2.1 Overall system design

Figure 1 shows a diagrammatic representation of the run-time components of this distributed network. In this diagram, jagged arrows denote communication between application client and server processes. Rounded, solid arrows show management communication taking place as an application server is instructed to checkpoint, and a Remex server is instructed to start a new process. The rounded, dashed arrow shows state being checkpointed by a checkpoint server.

User-application client and server processes may be resident on any machine in the network; client processes request that work be done on their behalf by server processes via remote procedure calls, and any results are returned from the server to the client.

The system design provides remote-execution facilities via remote-execution servers which reside on each site in the distributed network. These *Remex* servers allow processes to be started on any site in the distributed network, usually in response to a request from a management application.

At least one management client is provided by the checkpointing system, and runs on any site in the distributed network. The client (known as MiMan, for Migrator/Manager) allows human managers to interact with the distributed network, controlling many aspects of the execution of application processes, such as when they run, checkpoint and migrate.

At least one checkpoint server is provided by the checkpointing system, for each class of server in the distributed network. Checkpoint servers are responsible for collecting checkpointed state and either retaining it in memory or transferring it to stable storage. They are also responsible for returning captured state to a process that is undergoing restoration.

Several precompiler tools are provided as part of the high-level checkpointing system. These tools are bundled together into one package, which transparently adds all the required functionality to application process-code, allowing the compiled process to take part in checkpointing and migration at runtime.

3 DCE

A high-level approach to process checkpointing requires an underlying support environment that must include a naming service, a remote procedure call mechanism and universal type definitions. DCE is a collection of libraries, services and tools which is developed by the Open Software Foundation (OSF), which provides such an environment. In the implementation described here, state-transfer operations are implemented using DCE RPC.

The design of this high-level checkpointing scheme depends upon describing an execut-

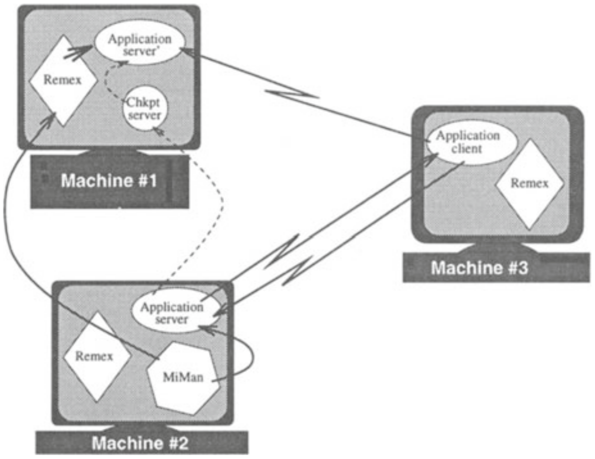


Figure 1 Diagrammatic snapshot of the runtime components of the HiCaM system.

ing process's state in terms of a high-level abstract machine. However, the types defined by high-level programming languages are often incompletely defined, making it impossible to fully describe the state of a process using these types. For example, the C programming language does not define what size an integer-type must be. Consequently, state which is defined using a C integer-type will have different meanings on different hardware platforms. DCE IDL types have the same definition on all machines. Since IDL type definitions are complete and consistent, interfaces defined using IDL types have the same semantics at any node in the system. Similarly, the state of a process can be defined using IDL-types, in which case the data which makes up a process-state will also be consistent at any node.

4 CONFORMIST SERVERS

In most distributed environments, applications are structured into *client* or *server* processes. Server processes perform tasks in response to a request by one or more client process. The request is delivered to the server via an RPC, and any results are returned to the client in the same way.

A user-application server requires additional functionality if it is to be able to take advantage of the high-level process checkpointing mechanism presented here. Additional functionality should be added transparently, and from the point of view of other application-processes, the resulting *conformist* server should be indistinguishable from the original server at runtime.

HiCaM includes precompiler tools which generate a server's checkpointing/migration functionality, based on the original server code. The checkpoint/migration code includes function and data structure definitions, as well as auxiliary files, which are compiled separately and linked to the general application files to produce a conformist server process. The precompiler tools which add the conformist functionality are bundled together and the applications programmer need only invoke them from the command line or a make-file. Figure 2 gives a diagrammatic summary of the functions that are added by the preprocessing tools in the current system implementation. In this diagram, the four T-shaped boxes represent the additional functionality that is added during pre-processing by the HiCaM code generator. The three T-shapes to the right represent interfaces: a general application will have its own interface(s), but additional interfaces are also required.

4.1 Thread Monitoring Capabilities

A multithreaded server may concurrently service the RPCs of many clients, and each concurrent thread of execution can alter the server's state. However, at checkpoint time, the server's state must be guaranteed to be stable; all the server state must be accessible at a high level of abstraction, and the state may not change while a checkpoint operation is taking place. Therefore, thread monitoring capabilities must be added which transparently allow other checkpoint/restore functions to determine the total number of threads present in a server at any time, and which threads are active (executing) at that time. In addition, the thread monitoring code must describe these aspects of the thread's execution in terms of the high-level abstract machine, since this is the level at which checkpointing will take place.

In the current system prototype, thread monitoring is implemented in the following way. Thread monitoring code is automatically generated, which replaces the manager-code address with the address of a new function, which in turn registers the addition of a new thread and identifies the client that is bound to that particular thread. The new function then passes the original RPC calling parameters to the conventional manager code, which then does the work that was requested by the client process. Any results from the manager code are passed back to a second precompiler-added function which deletes the current thread from the list of active threads (since the RPC has now effectively finished), and returns the result to the client as a conventional RPC return-value.

Additional consideration has been given to the common scenario, where many clients re-bind to a multithreaded server, after that server has migrated. If the server was in the process of servicing more than one RPC when it was migrated, it will almost always be necessary for each of the respective clients to not only rebind with that same server, but to rebind with the same thread within that server, since each thread will often contain information which is specific to the calling client.

4.2 Interface withdrawal and re-advertising functionality

In a general distributed environment, server processes interact with their clients through well-defined interfaces. When a server first begins execution, it advertises its availability by registering its interface with the name-service. However, if the server is later migrated to a different site, the server's name-service registration must be updated to reflect its new location; if the old binding information is not updated, subsequent clients will attempt to

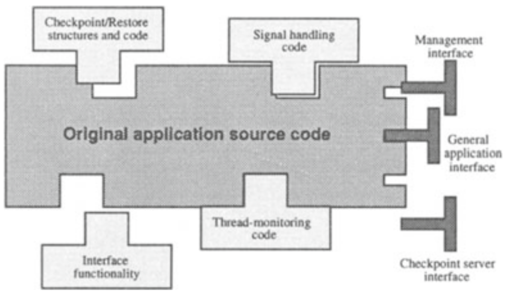

Checkpoint/Restore structures and code
Signal handling code
Management interface
Original application source code
General application interface
Thread-monitoring code
Interface functionality
Checkpoint server interface

Figure 2 Additional code added by HiCaM preprocessor tools.

bind with servers which do not exist at the expected (old) location, resulting in runtime errors. The system presented here includes functionality which withdraws the server's interface prior to a checkpoint operation, to ensure that no clients may make a request of the server during a checkpoint and thereby render the server's state invalid. In addition, the additional functionality updates the server's binding information in the name-service as part of the restoration process, in the event that the server is migrated after being checkpointed.

4.3 Signal-handling and cleanup facilities

In UNIX-style operating systems, processes often interact via a software-signal mechanism, which resembles a hardware interrupt. However, DCE takes control of the signal facilities so they cannot be used by programmers as they would in a non-DCE environment. Although DCE provides other process-control mechanisms such as exceptions, signals still appear to be preferable in some situations, especially for low-level process control. For this reason, the checkpointing design includes signal-handling facilities; developers may write DCE programs which use signal-handling facilities in the conventional way. These signal-handling facilities are provided in the form of a function library, which is linked to the general server-code at compile-time.

Signal handling facilities are incorporated into the prototype implementation, together with functionality that allows programmers to provide clean-up routines for their applications. DCE does not remove CDS entries when a process exits, and this can lead to out-of-date information being accessed by clients as they search for a suitable server. For this reason, application cleanup routines are very useful in helping maintain the CDS and thereby avoiding client runtime errors.

4.4 Checkpoint and restore functionality

The main goal of this work is to enable processes to be checkpointed and migrated. The checkpoint/restore functionality makes use of all the functional additions mentioned previously; thread monitoring is needed to ensure that the process state is stable before and during a checkpoint; interfaces are withdrawn while a process is checkpointed in order to ensure that no further (state-altering) RPCs are accepted; those same interfaces may be re-advertised if the process is migrated; signals are used by MiMan and Remex to control the execution of applications.

However, still more utility must be added to application objects if they are to be checkpointed and restored. Existing systems which operate in a similar environment such as DC++, Arjuna, Argus and ANSAware-based schemes require the object designer to specify the state which is to be saved during a checkpoint operation; checkpoint and restore operations must be defined as part of the object's interface. The object-designer or programmer must decide which variables comprise the state that is to be saved during a checkpoint operation. Based in this decision, they must then code-up the actual checkpoint and restore procedures, or at least provide the checkpoint and restore operation definitions.

These existing checkpointing schemes can be improved upon in a number of areas. Firstly, existing schemes require more work from the application designers and programmers since they must at least write the checkpoint and restore functions, both of which can be very complex where large applications are concerned. This extra work increases the risk that programming errors will be introduced, and such errors would be difficult to detect if they occurred at runtime, when a process attempted to checkpoint or restore its state.

Secondly, the level of granularity of these programmer-defined operations is necessarily coarse. A programmer cannot know which piece of code will be executing when a checkpoint request is received, so the state which can be checkpointed can include only global variables. Unless a checkpoint call is guaranteed to arrive while a certain function is executing, the variables which are defined locally to that function may not exist at checkpoint-time. Under such a scheme, either the application must be written with a lot of global state, or much information may be lost.

Multithreaded servers add to the local-state problem since the state of a multithreaded process is more difficult to describe at a high level of abstraction, than is the state of a single-threaded process. If the high-level checkpoint operation works only at a coarse level of granularity, then variables which are defined locally to functions are of no concern, as outlined above. However, if checkpointing operates at a medium-to-fine level of granularity, the state of execution of each individual thread must be described, since each one could be executing a different piece of code when a checkpoint request is received.

For these reasons, HiCaM includes preprocessor tools which incorporate a code-generator. The prototype generator, which is included as part of the overall current system implementation, produces all the code that is required to checkpoint and restore a general process, within the bounds that are outlined in section 9 of this paper. After the additional code has been generated, it is compiled and linked to the original application code, to produce a conformist executable file. The conformist process can be configured to checkpoint its state automatically at pre-defined intervals, as well as in response to a checkpoint request. The checkpoint and restore operations take place at a medium level of granularity, allowing executing threads to be checkpointed, so reducing the amount of information that is lost. Process state can include locally defined variables as well as global data, since the thread monitoring functionality can detect the stage of execution for each thread when a checkpoint request arrives. It is possible to checkpoint multithreaded servers under the current implementation.

4.5 Management Interface

A general object is transformed into a conformist one so that it can take advantage of certain management functions such as checkpointing and migration. For this to occur, the conformist object also requires a *management interface*, in addition to any other interfaces it may already possess. The management interface is defined in DCE IDL, and is transparently added to any other application-based interfaces by the system pre-compiler tools. Operations which are included in the management interface are outlined below.

- **startServer** Causes the server to begin execution. This operation would usually be called when an application is first brought into service. However, it may also be used to reinitialize a process.
- **stopServer** Used to shutdown an application. This function will usually invoke other cleanup functions, such as the one to remove the server's information from the CDS.
- **chkptServer** Used to capture the state of an application. In the current version of the system, this call trips the checkpoint timer, causing a checkpoint operation to occur and the timer to reset.
- **restrServer** This operation is called after a checkpoint or migration has occurred. It is used to inform the process that it should restore some old state (including UUID), rather than start again from scratch.

The management interface of every conformist application supports the same operations, allowing all conformist applications to be managed in the same way (by the MiMan tool, for example). A `stackFrameUnion_t_p` is a pointer to a data structure which stores the application-dependant state that is saved by a checkpoint operation. The exact contents of a `stackFrameUnion` depend on what stage of execution the process is at when a checkpoint request arrives so the `union-tag` is used to store this chronological information at checkpoint-time. By storing the state information in this way, it is possible to have just one checkpoint operation on every management interface, even though very different state-information will be saved, depending on what stage execution is at when the checkpoint call is received.

5 CONFORMIST CLIENT PROCESSES

General client applications require additional functionality if they are to take advantage of the checkpoint/migrate system at run-time. In order that the overall checkpointing problem could be simplified, this work has initially concentrated on servers, resulting in an initial, simplified design that includes stationary clients and migratory servers. Based on this strict client-server design, a general client only requires additional functionality which allows it to keep track of a migrating server.

It is common for a client to lose contact with a server due to process migration, especially if the RPC is long-running and the client is forced to wait some time for a reply. If a client-server binding is broken, the client will need to rebind to the same server in order to receive a meaningful reply.

In the current implementation, additional facilities are added to a client process by the HiCaM precompiler tools, allowing it to transparently detect a failure in the RPC connection after a predetermined timeout period. Depending on the type of communication error, the client is able to determine that its link to the server has probably been broken, and that it should try to rebind to that same server again. When the checkpoint is complete, the conformist server will readvertise its binding information, whether it has migrated or not.

The additional client-code allows the conformist client to transparently find the new location of the migrated server in the CDS, based on the server's UUID. The client then transparently rebinds to the server, and can thereby receive the correct RPC return-value when the server completes its RPC request. Current work includes the extending the code generator, so that there is less distinction between client and server processes, and applications which are both clients and servers can be checkpointed.

6 DESCRIPTION OF THE REMEX SERVER

As illustrated in figure 1, a Remex server resides on every site in the distributed network, providing remote process-execution facilities at its local site. Remex servers are transparent from the point of view of application developers and programmers and from the point of view of runtime application-objects themselves. The only entities which communicate directly with Remex servers are the MiMan migration/management applications, which are discussed in the next section.

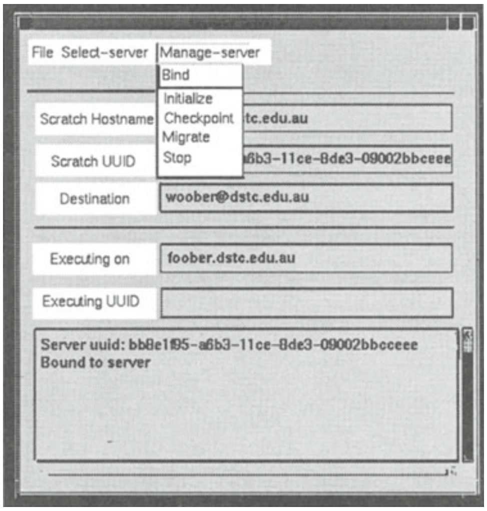
Remex servers provide a layering between the high-level checkpointing system and individual underlying operating systems. While all Remex servers advertise a standard interface, process instantiation requires calls to be made to the operating system of the machine on which the process is to begin execution. For this reason, Remex servers are written to include some platform-specific code. They do not migrate, but instead provide part of the fixed infrastructure which exists to support application migration at each site in the distributed network.

7 DESCRIPTION OF THE MIGRATION MANAGER

The MiMan migrator/management application provides the interface between human managers and the rest of the distributed computing network. Human managers can interact with the components of the system via a graphical user interface, which allows them to control aspects of a conformist server's execution including its checkpoint frequency and destination (migration) site. Figure 3 shows a copy of the MiMan user interface.

MiMan has five components: a target conformist-process selection component, a destination-site selection component, a management component, a GUI component, shown in figure 3, and interfaces for interacting with target conformist processes and Remex servers.

MiMan allows the user to select individual servers based on location and then on UUID.



Scratch Hostname: tc.edu.au
Scratch UUID: i6b3-11ce-8de3-09002bbceee
Destination: woober@dstc.edu.au
Executing on: foober.dstc.edu.au
Executing UUID:

```
Server uuid: bb8e1f95-a6b3-11ce-8de3-09002bbcceee
Bound to server
```

Figure 3 A screen-dump of the MiMan user interface.

This means that users can concentrate on certain sites of interest and can also choose between a number of servers if more than one is present. When a server has been selected, MiMan can bind to that server, making it the target of future management requests, until such time as another server is selected. In the current implementation, MiMan performs the operations on the servers that were discussed in the management interface item of section 4.5. In addition, MiMan allows the user to select a destination site, and to migrate the server-state to that site.

8 DESCRIPTION OF CHECKPOINT SERVERS

The final run-time component of HiCaM is the checkpoint-server. The responsibility of the checkpoint server is to interface with a checkpointing application and receive the captured process state. The checkpoint server is then responsible for retaining the state in memory or, more usually, for saving the state onto some form of stable storage. There are two advantages to this design. The first advantage is that the actual state-saving functionality is removed from the application, thereby improving its performance and reducing the amount of required conformist functionality. The second advantage is that the design of the stable-storage mechanism can be developed separately to the checkpoint and migration functionality.

The source-code for the checkpoint-server is generated by one of the precompiler tools, based on the checkpoint and restore code that is generated for an individual server. Each conformist server-application checkpoints its state to its matching checkpoint-server which has a compatible checkpoint and restore interface. From the point of view of general applications, checkpoint servers are transparent, forming part of theunderlying checkpointing system.

9 FUTURE WORK

Current research includes a study of how high-level checkpointing schemes affect the size of a general executable file and the efficiency of the executing process. The results of these experiments will be published at a later date.

In the short term, the functionality which allows clients to rebind with an individual server thread will be refined and extended. The aim is to allow clients to continue working within the same light-weight context, even if their multithreaded server should migrate during a task. HiCaM is also being extended to remove the strict boundaries between client and server applications, allowing clients to have the same functionality with regard to checkpointing and migration, as servers currently do. Further plans include research into non-object dependencies which often occur as part of process I/O.

10 SUMMARY

This paper has presented the design of a high-level checkpointing and migration system, HiCaM, which includes compile-time tools, management functionality and transparent runtime process support. The system provides a basis for the implementation of resilient processes and for process migration in heterogeneous distributed systems. The underlying checkpointing mechanism captures process state at a high-level of abstraction, allowing it to be meaningfully transferred to any site in a heterogeneous distributed system. In addition, the checkpointing mechanism is able to capture the process-state at a finer level of granularity than is possible with existing similar systems, can handle dynamically allocated memory, and is also able to checkpoint and restore multithreaded applications.

Client processes are able to continue working with migratory servers, transparently detecting the server's move and then tracking it to its destination site, where re-binding occurs.

Programmers are not required to learn any special languages or constructs to use this system, nor are they required to write any additional code; preprocessing tools are provided which transparently transform a user application into a *conformist* application. However, the server application must exist in a distributed environment which includes a naming service, universal types and an RPC mechanism.

HiCaM also provides underlying services which form a checkpointing/migration infrastructure. These services include Remex servers, checkpoint servers and MiMan client-applications. MiMan provides a graphical interface which allows human managers to interact with the distributed system, directing the execution of individual server-applications.

The system design presented in this paper has been implemented on a distributed network of DEC Alpha and IBM RS6000 machines, running OSF/1 and AIX respectively.

All machines in the distributed network also run OSF DCE. The number and type of platforms is expected to increase in the future.

ACKNOWLEDGEMENTS

The work reported in this paper has been partly funded by scholarships from the Department of Computer Science at the University of Queensland, and the CRC for Distributed Systems Research. The CRC for Distributed Systems Research is funded in part by the Cooperative Research Centres Program through the department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

The author would like to thank his supervisors, Dr. Kerry Raymond (DSTC/CiTR) and Prof. Andrew Lister (Dept. Computer Science, UQ), for their assistance and suggestions during the course of this research.

REFERENCES

Mishra, S. and Schlichting, R. (1992). Abstractions for Constructing Dependable Distributed Systems. *Technical Report. Department of Computer Science, University of Arizona*

Nuttall, M. (1994) A brief survey of systems providing process or object migration facilities. *ACM Operating Systems Review*, 64-80.

van Renesse, R., van Staveren, H. and Tanenbaum, A. (1988) Performance of the Worlds Fastest Distributed Operating System. *Operating Systems Review*, **22**, 25-34.

Bal, H. (1992) Fault-tolerant parallel programming in Argus. *Concurrency: Practice and Experience*, **4(1)**, 37-55.

Attardi, G. et. al. (1988) Techniques For Dynamic Software Migration. *ESPRIT 88: Proc. 5th Annual ESPRIT Conference*, 475-91.

Attardi, G. et. al. (1987) Specifications for High Level Abstract Common Machine Version 3.1. *Chameleon Report TR-87-40*, 1-62.

Attardi, G. et. al. (1987) Incremental Loading in HLACM. *Chameleon Note TR-87-38*, 1-9.

Shrivastava, G.D.S and Parrington, G. (1991) An Overview of the Arjuna Distributed Programming System. *IEEE Software*, 66-73.

Schill, A and Mock, M. (1993) DC++: Distributed Object-Oriented System Support on Top of OSF DCE. *Distributed Systems Engineering*, 112-25.

ANSAware. (1993) ANSAware 4.1 Application Programming in ANSAware. *Programming Manual.*

Redhead, T. (1995) Implementation of high-level checkpointing in heterogeneous distributed systems. *to be submitted.*

Shirley, J. (1992) Guide to Writing DCE Applications. *O'Reilly and Associates.*

Rosenberry, W. et. al. (1992) Understanding DCE. *O'Reilly and Associates.*

Theimer, M. and Hayes, B. (1991) Heterogeneous Process Migration by Recompilation. *Proceedings 11th Int. Conf. on Distributed Computing Systems.* 11-25.