JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 6, July-August 2007

Persistent Processes and Distribution in Timor

J. Leslie Keedy, Klaus Espenlaub, Christian Heinlein and Gisela Menger, University of Ulm, Germany

Abstract

The paper explains the concept of persistent processes and threads in Timor, showing how it is based on the in-process (procedure oriented) model of process execution. In their Timor form processes and threads can persist even when a user is logged out. They can also invoke remote persistent objects with the same semantics as invocation of local objects.

1 INTRODUCTION

A companion paper [11] describes how types defined in Timor can be instantiated as "file" objects which can subsequently be addressed using capabilities both from within Timor programs and at the operating system (OS) level. This paper builds on and extends the ideas presented in that paper, discussing the closely related themes of persistent processes and the Timor distribution concept. A knowledge of the companion paper is assumed, although relevant aspects are summarised in section 2.

Before the Timor facilities for persistent processes and threads are presented, section 3 distinguishes between two fundamentally different process structuring techniques, the in-process (procedure oriented) and out-of-process (message oriented) techniques, and shows that although these can be considered to be formally equivalent, the former has some significant practical advantages.

Section 4 describes the features which allow a normal Timor programmer to write parallel programs. Section 5 sketches out aspects of Timor which allow a system programmer to develop types and implementations which can enhance the Timor run-time environment.

Section 6 outlines the Timor concept for distributed systems. Section 7 briefly discusses implementation issues. Related work is described in section 8, and section 9 concludes the paper.

Cite this article as follows: J. Leslie Keedy, Klaus Espenlaub, Christian Heinlein and Gisela Menger: “Persistent Objects and Capabilities in Timor”, in *Journal of Object Technology*, vol. 6, no. 6, July-August 2007, pp. 91-108 http://www.jot.fm/issues/issue_2007_07/article2

2 THE TIMOR OBJECT MODEL

Timor distinguishes between type definitions and their implementations [7-9]. A Timor type definition (introduced by the keyword `type`) can have several implementations (introduced by the keyword `impl`) and different instances of the same type can have different implementations, even within a single program.

Type Definitions

A type definition potentially consists of instance methods, abstract variables and makers (constructors). Instance methods operate on a particular instance of the type. They are designated either as `op` (for operation) or `enq` (for enquiry). An `op` method can modify the state of its instance, an `enq` can only read the state. Abstract variables superficially resemble public fields. However these are actually pairs of instance methods (an `op` and an `enq`) which by default set and get the value of an internal variable, but which can be programmed to have other implementations [9]. Thus Timor types fully conform to the information hiding principle. Here is a simple example of a type definition:

```
type Person {
instance:
 String name, address;  // abstract variables
 Date dateOfBirth;
 enq int currentAge();
}
```

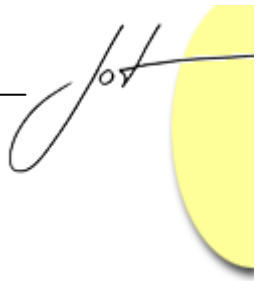
To avoid problems with binary instance methods (see [1]) instance methods may not have parameters of the type being defined.

Implementations

A type can have several implementations. An implementation has its own identifier and names the type which it implements, e.g.

```
impl PersonImpl of Person {
state:                  // the state variables of an instance
 String name, address; // concrete variables
 Date dateOfBirth;
instance:               // the instance methods
 enq int currentAge() {/* code to calculate current age*/}
}
```

An implementation of the method pair corresponding to each abstract variable is automatically added by the compiler, unless the programmer explicitly adds code, e.g. the abstract variable `String name` has the code:



```
op String name(String name) {return this.name = name;}
enq String name() {return this.name;}
```

Files, Local Objects and Values

An instance of a type can serve many different purposes, e.g. as a data abstraction, as a subroutine library, as a program, and even as a file which is structured according to the information hiding principle. In support of this flexible approach, a type can be instantiated at any of three levels: (a) as a *file object*, (b) as a *local object*, or (c) as a *value* within an object.

A *file object* is an independent persistent entity which can be accessed - within a Timor program and/or at the operating system (OS) level – only by presenting a *capability*, which serves as a reference to the file and defines which methods of the object the holder can invoke. File objects serve a purpose similar to that of conventional files and programs in an OS environment, but can also be used e.g. as the equivalent of remote objects in Java.

A *local object* is an independent sub-object within a file (or process) object. In Timor local objects and values can exist only within files and processes. Hence the equivalent of a conventional OO program is a Timor file object in which local objects are created, and with one or more methods containing code equivalent to the code of a conventional program. A local object can be accessed only via a *reference*, which is in effect a local capability that cannot be released from the context of its file. Possession of references with appropriate access rights determines which objects in the file have what method access to a local object.

A *value* is an instance of a type which is accessible only to the (file or local) object in which it is embedded, i.e. it is a private instance which is a part of another object. The *state* of an object consists of its values, which in accordance with the information hiding principle are visible or accessible only to the object in which these values are embedded. Capabilities and references can also be stored in the state of an object. Values (including reference and capability values) can also be instantiated locally in a method, in which case they reside on the current execution stack.

Instantiating Types

When a type is instantiated the maker (a constructor) always formally returns a *value* of that type. There is a `new` operator which can transform a value into a local object, and a `create` operator which can transform a value into a file object.

A value belongs to only one object and direct access to a value can only occur from within that object[1]. A value can only be assigned to a value variable of the type (or a supertype thereof). Given a type `Person` an actual `Person` value can be instantiated in a statement such as:

[1] There is no operator in Timor which can create a pointer or reference from a value, and values cannot be passed by reference to methods.

```
Person aPersonValue = Person.init();
```

where `Person.init()` is a maker invocation.

Local objects correspond loosely to the usual notion of objects in OO programs. However, whereas in a language such as Java or C++ such objects must (where appropriate) be explicitly made persistent[2], in Timor local objects are naturally persistent unless they are explicitly or implicitly deleted, because they are always embedded in file objects.

The `new` operator accepts a value (which may or may not have just been instantiated by a maker), creates a local object the state of which is a copy[3] of the value, enters this into a local object table for the current file, and returns a reference to the new object. This reference can be assigned to a reference variable of the same type (or a supertype thereof), as the following example illustrates:

```
Person* aPersonRef = new Person.init();
```

Similarly the `create` operator accepts a value, creates a file object with a state which is a copy of this value and returns a capability for the new file object, which can be assigned to a capability variable, e.g.

```
Person** aPersonCap = create Person.init();
```

File objects are accessed via capabilities. Like files in conventional systems file objects are known at the OS level. However, they differ substantially from conventional files in that they are objects which – like normal OO objects - can only be invoked via their associated methods. Such invocations can occur both at the OS level and from within Timor programs.

Relationship between Value Variables, References and Capabilities

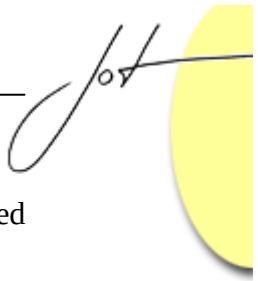
For each type a *type handle* automatically exists. This serves as a supertype for capabilities, references and values of the type. It is designated using a three star notation (e.g. for a type `T` it is written as `T***`). This makes it possible to write programs which can for example receive parameters of a specified type without defining exactly which mode is to be passed. (A type safe cast statement can be used to determine the actual mode of the entity assigned to the type handle.)

Programs

There is no special concept of a "program". A conventional program can be programmed as a normal instance method of a normal type, which can be instantiated as a value, a

[2] typically by using libraries, as in C++, possibly assisted by further mechanisms, e.g. the Java `Serializable` feature.

[3] Although `new` formally copies the value, the compiler optimises away redundant copying.



local object or a file object. If instantiated as a file object the "program" can be invoked via a capability (either from within a Timor program or at the OS level).

Because a program is simply an instance method, types can be defined which in effect are collections of programs, e.g.

```
type GamesCompendium {
instance:
 op void chess();
 op void draughts();
}
```

Defining programs as normal instance methods means that they do not have a special name (such as "main"). An interesting effect of this approach is that programs can be executed internally in the context of other programs. For example the chess program can be executed within some other program or object with an expression such as:

```
GamesCompendium.init().chess();
```

In this case a new "value" is created for the duration of the program execution. In principle it is also possible to execute a program as a local object, e.g.

```
(new GamesCompendium.init()).chess();
```

Provided that the state left behind by the method is not subsequently assigned to a variable, the difference is uninteresting from an application viewpoint.

3 PROCESS EXECUTION MODELS

To clarify our terminology we define a *process* as a static representation of an independent activity and a *thread* as a unit within a process which can actually carry out a computation (using a separate stack). A process can contain multiple threads which might be introduced to simplify the logic of a process and/or to improve efficiency by allowing the activity to be carried out in parallel.

There are two fundamentally different models for mapping processes and threads onto objects. In the following description of these we assume initially that a process has only one thread. Multithreading is discussed as an extension to these models.

Alternative Execution Models

The first model assumes that there is a fixed (one to one) relationship between an object and a process, such that the code of an object is executed by its own thread. This corresponds to the typical client-server model. To request a service, a thread of a client object sends a message to the process of a server object. When the service has been carried out the server typically sends a reply in the form of a message to the client process. This model is frequently used at the operating system level, for communication between application programs and OS modules, between OS modules themselves and (because the OS provides no alternative) between application objects. It is for example the underlying model used for communication between remote objects in Java (although the programmer is given the impression that he is making a remote procedure call).

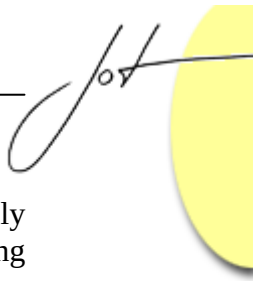
In this model each object has an input buffer of message requests. If an object provides a number of services, each different type of message can represent a different service provided by the server. The thread associated with an object in principle executes an infinite loop in which messages are taken from its input buffer and are processed in turn, with replies being passed back to the sending processes as messages. If an object's input buffer is empty, the thread waits until a new message arrives. At the level of OS modules, Lauer and Needham [13] called this the *message oriented* model. It is also known as the *out-of-process* model (e.g. [14]), because services are handled out of the user process (i.e. in another process).

In the alternative model (which Lauer and Needham called the *procedure oriented* model, also known as the *in-process* model) each application program is represented by a separate process, but there is no fixed relationship between a process and a server object. Instead of sending messages to other objects, an application program invokes the services of other objects by calling these as methods, without switching threads. Each public method of an object represents a separate service provided by that object. When the handling of the service completes, the method exits and execution is resumed in the calling object, without switching threads. In this model processes do not "belong to" objects. Instead, a thread moves from one object to another as services are requested and are completed, in a stack-like way. This model corresponds to the way the internal objects of an OO program communicate with each other.

Lauer and Needham considered these two models of process execution to be duals of each other and thus fully equivalent. While this view may be theoretically correct, Ramamohanarao [14] has shown that there are many practical differences, which on analysis favour the procedure oriented model. In the following subsections we briefly explain some of these.

Parallelism via Threads

Parallel activity in an object can be achieved in the out-of-process model by associating multiple threads with a process, i.e. with an object. If this is done statically then the existence of extra threads is only useful to the extent that multiple messages which can be



processed in parallel are actually (dynamically) present. Otherwise the threads simply wait for messages and increase the process scheduler overhead without improving parallelism. If, on the other hand, threads are created dynamically on a need basis, a thread creation overhead arises whenever a thread is dynamically created.

Parallelism is achieved in the in-process model in that the user application explicitly creates multiple threads for each appropriate parallel user activity. Each thread can execute independently of the others and potentially in parallel with them. Each can move from object to object by invoking their methods as procedure calls. Such parallelism reflects the parallel activities of a user process, not merely attempts to optimise system efficiency by adding more threads to an object, i.e. threads are created on a genuine need basis, and if multiple users or user activities need parallel access to an external service, no new threads need be created and no processes need to hang around waiting for messages. Hence the number of threads in an in-process system represents the number of genuinely needed, user-initiated[4] activities.

In the in-process multithreading model there is no process scheduler overhead due to the creation of extra threads "in case" they are needed, nor is there an extra thread creation cost when multiple users happen to invoke the same object. Furthermore conventional parameter passing is not only more efficient but also much more flexible than passing information via message buffers, and when a client invokes a server, there is no process switching overhead as such (although a part of the overhead associated with a process switch can occur, to switch to a new addressing context). Thus an in-process invocation of a server object is always cheaper than the out-of-process equivalent, because process scheduling is not involved.

Synchronisation

In a basic out-of-process system, in which there is only one thread associated with each object, the programmer does not need explicitly to include synchronisation code, because the synchronisation is organised by the message passing system, e.g. by treating message buffers as bounded buffers [3]. However, as soon as extra threads are added to a server object in an attempt to increase parallelism, explicit synchronisation has to be added, to ensure that the internal data structures of the object remain consistent. In other words synchronisation occurs both on the message buffers and on the server's private state data. Although the latter can be organised using monitors which enforce mutual exclusion [5], significantly more parallelism is often achievable if reader-writer synchronisation (cf. [2]) is used.

In an in-process system synchronisation is never needed on the "message passing" operation, as this takes the form of parameter passing on separate stacks for different threads. However, it is always possible that multiple threads can be concurrently active in the same object, making explicit synchronisation necessary at the level of the private state data of the object.

[4] If a server object can benefit from parallelism, it can of course create multiple threads dynamically in the in-process model.

Hence although a basic single-thread out-of-process system has the advantage of implicit synchronisation, an in-process system designed to achieve a high degree of parallelism requires the same explicit synchronisation as its out-of-process counterpart but has the advantage that bounded buffer synchronisation (which in a highly parallel system can easily become a bottleneck) is not needed.

Identifying the Initiator of an Activity

There are many good reasons for wishing to identify who is the initiator of some activity which may actually be carried out in some service (e.g. OS) object. One example is the need to charge users for their use of services. In the in-process model this is easily (and cheaply) done, simply by examining the identity of the process carrying out the activity (because the process does not change when a server object is invoked). This point is also relevant for monitoring activities. But the most significant advantage of the in-process model in this respect relates to protection.

In many out-of-process systems the question which user is attempting to carry out some protected activity cannot easily or cheaply be answered, because the active process is often a system process (i.e. a process associated with the current server object rather than with the current user or application). It is of course possible to organise an out-of-process system such that originators of messages can be traced, but in practice such a feature is rarely provided as a general facility, because it would be relatively expensive to implement.

Choice of Model for Timor

Other programming languages, such as Java, have not made an explicit choice between these two models. Instead the tendency is implicitly to follow the in-process model for interactions between local objects within a single OO program, but for interactions between independent major objects (e.g. remote objects in Java) to accept the limitations imposed by the (out-of-process) operating system environments in which they are used, with the result that the semantics for object interactions differ between these two levels (see section 8).

In designing Timor we have decided in favour of a consistent model with uniform semantics. In view of the advantages of the in-process model which have been outlined in this section, Timor rigorously supports the in-process model. This provides Timor with a significant protection (and monitoring) advantage over other OO programming languages [12]. However, for those convinced of the advantages of the out-of-process model, Timor is not a hindrance, because it is easy to simulate the out-of-process model in an in-process system. (Simulating the in-process model in an out-of-process model is much more difficult.)

4 PARALLEL TIMOR PROGRAMS

This section introduces those features of Timor which are needed to write normal parallel programs. As in other programming languages the programmer can always assume that when his code starts executing in one of the interface methods of an object an already existing thread executes the code which he writes. This is the thread which invokes the method of the object being programmed.

Creating Threads

To provide parallel activity for the execution of the code of an object, the programmer can create new threads by means of the `thread` operator. This returns a thread reference (of type `Thread*`)[5]. The `thread` operator is followed by a method call, which specifies the code that is subsequently executed as an independent thread.

To create a new thread, a statement such as the following can be used:

```
Thread* aThread = thread x.m(a, b, c);
```

The parameters of the method (here method `m` of object `x`) are evaluated in the *creating* thread and are made available as values, references[6] and/or capabilities to the new thread. If the method is defined to return a result, this is discarded, i.e. the new thread cannot communicate a result to its creator by returning a value. Results can of course be returned in that the creating thread passes a reference or capability for an object as a parameter to the new thread, which then writes values to be returned into this object.

Thread References

A thread reference is returned when a new thread is created and it can be used by the creating thread to control the further progress of the new thread, e.g. using methods such as `suspend`, `continue`, `delete` of the type `Thread`.

Conceptually a thread reference is like any other reference. It can be assigned to variables of the type `Thread*`, and multiple references can concurrently exist for a thread. A thread reference cannot escape from the file object in which it was created, because, like other references, it cannot be passed as a parameter nor returned on inter-file object method invocations (see [11]).

[5] Using the `thread` construct, a programmer cannot create values or capabilities of type `Thread`. We shall see in section 5 how thread capabilities can be created. Thread values can never be explicitly created.
[6] Whether such parameters can actually include references depends whether `x` is a file object or not (see [11]).

Inter-File Communication

Although thread references cannot escape from the file object in which they are created, the threads themselves are not confined to executing within a single file object. In accordance with the in-process model, a thread can visit different file objects by invoking their methods (assuming that it has access to capabilities with appropriate access rights).

Synchronisation

As indicated in section 3, explicit synchronisation of the state data of an object is always necessary in an in-process system. Nevertheless, synchronisation as such is not part of the Timor language, in contrast with Java. However, there is a basic type `Semaphore`, which is part of the language definition.

The normal programmer does not have to explicitly program with semaphores. Instead he can for example easily achieve mutual exclusion or reader-writer synchronisation simply by associating general purpose synchronising qualifiers with the objects to be synchronised (at both the local and file object levels [10]).

Explicit coordination of the cooperating threads of a process can also be achieved by semaphores (e.g. used as private semaphores) and if appropriate local or file objects can be used as bounded buffers for exchanging information [3].

Processes, like file objects, have world-wide unique identifiers which are not re-used over time. Each thread has an identifier which consists of the process number and a suffix (which is unique within the process). The initial thread of a process is always thread 0. There is a pseudo variable `processNum` which allows the process number of the currently active thread to be obtained, and a similar pseudo variable `threadNum` for obtaining the current thread number (within its process). In this way communicating threads can identify each other.

Thread Deletion

A thread persists either until it is explicitly deleted (using a reference) or until it exits from the method in which it started executing. In addition all the threads created within a particular file object (i.e. where their thread references are stored) are deleted when the file object is deleted. All the threads belonging to a process are deleted when the process is deleted.

Deleting a thread has no effect on other threads which it may have created, or which may have created it. This is possible because there is no implicitly declared data which a thread automatically shares with its parent or children threads that is automatically deleted when a thread terminates.

5 THE TIMOR PROCESS MODEL

A normal *application* programmer who wishes to write parallel programs need understand only a straightforward thread concept and where appropriate synchronisation qualifiers, as discussed in the last section. However, a *system* programmer can use further features to implement types to create and/or manage processes for the *SPEEDOS* operating system[7] [4] or for the *SPEEDOS* environment used by Timor compilers in more conventional operating systems [11]. These features are now briefly outlined.

Creating Processes

In OO languages such as Java the initial thread associated with a program execution is created when the program is started. This is not normally the case in Timor, which automatically supports the concept of persistent processes [6].

Just like a file object in Timor [11], a process is a persistent object (which can be visible both at the OS level and the programming language level). A persistent process can loosely be thought of as a persistent file object in which the linkage stacks for inter-file object calls made by its threads are stored (and are used when the threads are active).

Each process object and each file object has an owner, in the sense of an externally identifiable user, and the unique identifier of this owner is recorded as part of the object[8]. Normal file objects created by a process have their ownership identifier set to that of the owner of the creating process. In this way it is always possible to identify the owner of a process and the owner of a file object. Executing code can obtain the owner identifier of the objects in his current environment, including the owner of the current process and current file. This can be used for example for carrying out protection checks (see [12]).

Timor supports a predefined type `Process`, which for technical reasons can only be instantiated as a file object. Two makers are provided to create a new process object. The first of these creates an additional process for the owner of the currently active process, e.g.

```
Process** myProcessCap = create Process.init();
```

The effect of this invocation is that an "empty" process object is created, i.e. without threads. The owner of this is marked as the owner of the creating process.

The second maker, called `newUser`, creates an "empty" process object for a new user, i.e. a new owner identifier is allocated for the new process:

```
Process** myProcessCap = create Process.newUser();
```

[7] www.speedos-security.org
[8] One key reason for this is that it supports a number of protection facilities [12]

Creating External Threads

A process is only useful if it contains threads which can be activated. In Timor there are two kinds of threads: *internal* threads for executing parallel programs, and *external* threads, which are responsible for initiating activities for the user, executing sequential programs and initiating parallel programs. As was described in section 4 internal threads are addressed and controlled by references, and therefore provide parallel activities for the methods associated with a particular file object (although they can visit other file objects), while an external thread represents an independent user activity associated with a process object which is addressed and controlled via capabilities. Hence an external thread is visible globally (e.g. at the OS level) and its capabilities can also be passed from one Timor file object to another.

A process has at most one external thread, which a user creates via the `createThread` method of the process object with which it is to be associated:

```
Thread** firstThread =
      myProcessCap.createThread(startObject, method, param);
```

If the parameters are invalid, this method can return the value `null`. The `startObject` is a capability for the file object in which the new thread starts to execute[9]. The `method` parameter is of type `int`, and defines the number of the method in which the thread starts executing[10] and the `param` parameter is a capability[9] which defines the addressing environment of the new thread. (For example it can be a capability for a directory – typically mapping string names to capabilities - which forms the root of the directory system available to the new thread.)

The effect of invoking `createThread` is that space for a linkage stack is made available in the process object (see below), the new thread begins executing in the defined method of the start object, and this method receives the `param` capability as its only parameter, i.e. only methods which are defined to accept a single capability as a parameter can be started by `createThread`, and any value which they return is lost. These are not serious limitations, since threads created in this way are special, as we shall now see.

Deleting Processes and External Threads

The basic idea behind a persistent process is that it exists, and can contain persistent threads, over a potentially long period of time. In accordance with the in-process model, its existence is independent of the existence of any other object (file, program, etc.). Under normal circumstances it is deleted only when an explicit decision is made (by invoking the `delete` method on the *process* capability).

[9] Its formal type is a `Handle`, since capabilities can be passed for different types of objects.
[10] At the *SPEEDOS* level the public methods of a file object are numbered, starting at zero. Further objects can be defined which map these numbers to the names used in the source code of a type.

An external thread (like an internal thread) is automatically deleted if it exits from the method in which it was created. An external thread can also be deleted if the `delete` method of its *thread* capability is invoked. The deletion of an external thread does not result in the deletion of its process, even if there are no further threads in the process, i.e. a process can be reused by creating a new thread using the `createThread` method.

Long Suspending External Threads

The idea behind persistent processes is that not only the process, viewed as a storage unit, but also its threads, as active computational units, can persist over long periods. For this purpose the SPEEDOS run-time environment provides a "long suspend" facility, which allows an external thread to voluntarily suspend itself for a long period (e.g. corresponding to the time a user wishes to log out or to suspend the execution of a particular program). A thread in the long suspended state does not need to be registered with the system's process/thread scheduler.

Long suspending can take two forms. An external thread can simply deregister itself, without affecting the execution of its internal threads, i.e. the latter can continue processing. This option is useful, for example, when a user wishes to log out but would like programs which he has started to continue executing in his absence.

Alternatively a long suspending external thread can request that the execution of its internal threads are also frozen[11]. This alternative can be useful for example simply to freeze the execution of a program which may not be executing as expected (e.g. because it appears to be executing in an infinite loop), in preparation for debugging, and it can be used to bring activity to a stop before the system is shut down.

Protecting Persistent Processes and their Threads

When the owner of an external thread decides to log out (or to suspend the execution of one of his programs), the Command Language Interpreter (CLI) - or any other file object which accepts a deactivation command - could in principle directly invoke the long suspend facility. However, to protect his objects and processes the CLI and other objects which accept a logout command should treat this as a signal to invoke a logout method of an *authentication* object, which then calls the long suspend method supported by the Timor run-time environment.

The effect of this is that when the user indicates that he wants to continue, usually via a login command to the Timor run-time environment, execution of the thread resumes at the statement following the long suspend call, i.e. in the logout method of the authentication object. The authentication object can then carry out arbitrary checks defined by the owner of the process to ensure that the user now attempting to activate the process is who he claims to be. Since users can select their own authentication objects, a

[11] In this case measures might need to be taken to avoid locking out threads from other processes which might be concurrently active in a file. The nature of these measures depends on the environment in which the threads are working. For example in a database system a roll-back may be appropriate, in a simple semaphore environment an exception might be raised which causes semaphores to be released, etc.

hacker has no *a priori* knowledge of how he can imitate the user (because there is no centrally defined authentication mechanism – such as a password system – and no centrally stored authentication information – such as a password file).

A simple modification of this procedure can be used to protect newly created external threads: when a new thread is activated, as described above, its start object might be defined to be a CLI object in a method which immediately calls the user's authentication object. The effect will be that the new thread enters the logged out state and can remain dormant until its owner, possibly a new user, first activates it.

6 DISTRIBUTION

The Timor distribution concept is in principle[12] very simple: Timor *file* and *process objects* can be located on *any* computer which has the necessary run-time software support. File objects can be accessed independently of their location, provided that an access path between cooperating systems exists (e.g. via the Internet).

In this context "access" to a file object means that one of its methods is invoked by a Timor thread currently active in some other file object, which may be stored on the same computer or some other computer in the network. The prerequisite for making a remote call (like the invocation of a file method of a file object on the same computer node) is the possession of a capability (with appropriate access rights) for the remote file. The location of a file can remain completely invisible to the holder of a capability.

In contrast with Java, for example, the semantics of invoking a remote object and a local object do not change. There are no features in Timor for serialising objects passed as parameters, and Timor does not have inconsistent semantics with respect to the passing of parameters (see the discussion in section 6 of [11]).

At the level of Timor programs and files, inter-file method invocations are the *only* means of communication between Timor file objects. There are neither message passing primitives nor protocols, although these can easily be simulated via user level file objects with appropriate methods (e.g. corresponding to bounded buffers).

Library routines can of course be developed which allow Timor programs to communicate with non-Timor programs and data entities which may be on remote (or local) nodes, but these are not relevant to the discussion in this paper.

[12] In practice communication errors can occur in a distributed system (e.g. when the destination computer cannot be reached, or when it fails to provide a response after acknowledging the receipt of a method invocation). Handling such cases cleanly implies– as in other systems - that appropriate exceptions must be provided and that recovery software can be provided in libraries, etc.

7 IMPLEMENTATION ISSUES

As was described in [11], the basic assumption in Timor is that objects are persistent. This applies equally to processes. All processes persist until they are explicitly or implicitly deleted. To this end Timor assumes not only a normal run-time environment (as for other programming languages), but also a more extensive run-time environment, in the form of an emulator for relevant parts of the *SPEEDOS* system (or better still, a run-time environment which is an actual *SPEEDOS* system).

The key features of a *SPEEDOS* emulator with respect to persistent processes include the use of

- memory mapped files as containers for file and process objects, thus avoiding the need to "serialise" data structures,
- a mechanism which allows threads to invoke the methods of local and remote file objects in a uniform manner, and
- the provision of a method for suspending processes at log out, thus allowing the process to carry out its own authentication checks (see section 5).

Persistent Process and Stack Organisation

When a new Timor process is created, a new persistent container (in a non-SPEEDOS environment a mapped file) is allocated at the node on which the creation operation takes place. Any threads subsequently created for this process are stored in this container. For each thread there is a stack of linkage information defining the inter-file calls made by the thread. From this linkage stack the current "location" of the thread (i.e. the file object in which it is currently active) can be established at any time.

For managing local calls within a file object (i.e. calls to local objects or to internal methods of an object) the language run-time system organises a local call stack which does not affect the process object, but which refers back to the process object of the thread in question, thus allowing the process container of each thread active in the file object to be located.

This organisation not only gives the compiler freedom to organise stacks, but it also solves the following problem which can occur in persistent systems.

Suppose that, as is frequently the case in practice, a method of a file object (or one of its local objects) has the task of creating a new local object – say a new bank account in a banking system – then the programmer will typically produce code along the following lines:

```
impl BankAccountsImpl of BankAccounts {
state:
 List<BankAccount*> accountsList =
                              List<BankAccount*>.init();
instance:
 op void createNewAccount(...) {
```

```
    BankAccount* newAccount = new BankAccount.init(...);
    accountsList.insert(newAccount);
  }
```

If the local stack frames of threads were stored in the process object the effect of executing such a statement would be that the new item (here `newAccount`) would be created in (a heap or stack of) the process object, and inserting it into the main persistent structure (in the file object) would result in a copy operation, which for large items would be unacceptable. By defining that the stack frames of a thread are stored in the container of the file object whose method is being executed, locally instantiated objects are created in the heap of the file object itself, such a copy operation can be avoided. (Note also that references between file objects are not permitted, as is explained in [11], so that a solution which involves instantiating an item in the file heap but referencing it from a stack in the process object is also excluded.) From the local stack of a thread (in the file object) it is possible to establish the identity of the process object to which the thread belongs.

Implementing Processes in Distributed Systems

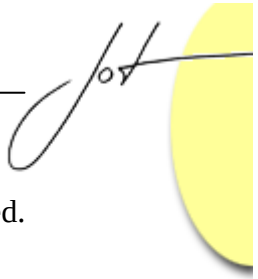
Capabilities can be used to invoke methods of files irrespective of their location. Hence a thread active in a file on one node of a network can simply invoke a method of a file on a different node (assuming that it has a valid capability for the remote file which provides access to the method requested).

In principle there are two ways of managing this situation: the data can be brought to the process, e.g. as in distributed virtual memory (DSM) implementations, or the process can be taken to the data, as for remote procedure calls (RPC). In genuine SPEEDOS systems a DSM implementation is the norm, although an RPC implementation can be easily organised. In the context of a SPEEDOS emulator for a conventional system, where paging across networks is not the norm, the more efficient implementation is a form of RPC, i.e. the process is taken to the data. This functions as follows.

When the first invocation of a file object by a thread of a process on a remote node[13] occurs, the SPEEDOS emulator on that node creates a proxy process container for the invoking process. The linkage data for the thread held in the process object of the calling node is updated to indicate that the thread has been transferred to the destination node. At the destination node the proxy process object is updated to reflect the origin of the call for the current thread, which retains the same identity as it had at the calling node. The *SPEEDOS* emulator organises that parameters (which can only be values or capabilities, cf. [11]) are copied to the proxy container. Thereafter the process can proceed in the appropriate file environment at the new node as if it were in the old node.

When an inter-file return operation is invoked, this process is reversed, i.e. return values are passed back to the calling node, the linkage information for the thread is

[13] Invocations of "remote" objects (in the Java sense) on the same node are not special for Timor, as the process object exists independently of a particular file object on the same node.



deleted and if this is the last thread in the proxy process container, the latter is deleted. Control is then returned, as usual, to the caller in the calling node.

8 RELATED WORK

The most important features of Timor persistence which find parallels in other OO programming languages (especially Java) were discussed in [11]. It is appropriate here to add that Java and other OO languages have no notion of a persistent process nor do they genuinely support the in-process process structuring model, despite attempts to give this appearance.

This is particularly clear in relation to remote method invocation. Because objects in Java are not naturally persistent, the destination instance of a remote method invocation must first be activated before it can be invoked remotely, i.e. remote objects, unlike Timor files, are active entities with their own thread(s). Hence under the surface the out-of-process model is used for communicating between remote objects in Java (unlike Timor), even though the programmer is given the impression that he is using the in-process model (as in the case of invocations between methods of local objects within a single program).

One consequence of this distinction is that the protection advantages of the in-process model discussed in [12] are not readily available in Java (and could not be made readily available without considerable programming effort and run-time cost). In particular Java does not (and could not, without considerable run-time overhead) provide a mechanism which uniquely identifies the user associated with an active process (in contrast with Timor, see section 3). This is unfortunate because this feature provides the basis for several significant protection possibilities, especially in conjunction with qualifying types and bracket methods (e.g. access control lists and revocation lists).

Finally, it is at least as easy in Timor to apply reader-writer synchronisation to an object (file or local) using a reader-writer qualifier [10] as it is to use a monitor or equivalent in a language such as Java, and the effect is greater parallelism.

9 CONCLUSION

A companion paper [11] presented a new concept for supporting *persistent objects* in programming languages, which inter alia allows a straightforward implementation of persistent database systems. This paper has extended the idea by describing how *persistent processes* can also be defined and implemented.

The Timor concept of persistent processes has a significant advantage over process concepts of other programming languages (leaving aside the protection advantages discussed in [12]), viz. distribution is genuinely invisible to the user (unless he specifically wants to know about the locations of his objects) without affecting the semantics of the language.

REFERENCES

[1] K. B. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. T. Leavens, and B. Pierce, "On Binary Methods", *Theory and Practice of Object Systems*, vol. 1, no. 3, pp. 221-242, 1995.

[2] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent Control with Readers and Writers," *Communications of the ACM*, vol. 14, no. 10, pp. 667-668, 1971.

[3] E. W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages*, E. Genuys, Ed.: Academic Press, 1968, pp. 43-112.

[4] K. Espenlaub, "Design of the *SPEEDOS* Operating System Kernel," PhD Thesis, *Department of Computer Structures*, University of Ulm, 2005.

[5] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549-557, 1974.

[6] J. L. Keedy and K. Vosseberg, "Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System," Proceedings of the 25th Hawaii International Conference on System Sciences, 1992, vol. 1, pp. 747-756.

[7] J. L. Keedy, G. Menger, and C. Heinlein, "Support for Subtyping and Code Re-use in Timor," 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, 2002, Conferences in Research and Practice in Information Technology, vol. 10, pp. 35-43.

[8] J. L. Keedy, G. Menger, and C. Heinlein, "Inheriting from a Common Abstract Ancestor in Timor," Journal of Object Technology, vol. 1, no. 1, pp. 81-106, www.jot.fm/issues/issue_2002_05/article2, 2002.

[9] J. L. Keedy, G. Menger, and C. Heinlein, "Taking Information Hiding Seriously in an Object Oriented Context," Net.ObjectDays, Erfurt, Germany, 2003, pp. 51-65.

[10] J. L. Keedy, G. Menger, C. Heinlein, and F. Henskens, "Qualifying Types Illustrated by Synchronisation Examples," in *Objects, Components, Architectures, Services and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany*, vol. LNCS 2591, M. Aksit, M. Mezini, and R. Unland, Eds.: Springer, 2003, pp. 330-344, http://link.springer.de/link/service/series/0558/papers/2591/25910330.pdf.

[11] J. L. Keedy, K. Espenlaub, C. Heinlein, and G. Menger, "Persistent Objects and Capabilities in Timor," in Journal of Object Technology, vol. 6, no. 4, May-June 2007, pp. 103-123 http://www.jot.fm/issues/issue_2007_05/article3

[12] J. L. Keedy, K. Espenlaub, C. Heinlein, and G. Menger, "Security and Protection in Timor Programs," *(submitted for publication)*, 2006.

[13] H. C. Lauer and R. M. Needham, "On the Duality of Operating System Structures," *ACM Operating Systems Review*, vol. 13, no. 2, pp. 3-19, 1979.

[14] K. Ramamohanarao, "A New Model for Job Management Systems", PhD Thesis, *Department of Computer Science*, Monash University, 1980.

About the authors

J. Leslie Keedy retired from the position of Professor and Head, Department of Computer Structures, University of Ulm, Germany in 2005, where he previously led the Timor language design and the Speedos operating system design groups. His email address is keedy@jlkeedy.net. His biography can be visited at http://www.jlkeedy.net/biography_short.php

Klaus Espenlaub completed his Ph.D. in Computer Science at the University of Ulm in 2005. He is currently employed by InnoTek Systemberatung GmbH. His research interests include secure operating systems, protection mechanisms and computer architecture. His email address is klaus@espenlaub.com.

Christian Heinlein has been working as a Scientific Assistant at the University of Ulm, Germany, where he conducted the research project APPLEs, that aims at developing "Advanced Procedural Programming Languages," which are both conceptually simpler and more flexible than standard object-oriented languages. He can be reached at christian.heinlein@uni-ulm.de. See also www.informatik.uni-ulm.de/rs/mitarbeiter/ch/apples.

Gisela Menger received a Ph.D. in Computer Science from the University of Ulm in 2000. She recently retired from the Department of Computer Structures at the University of Ulm. Her research interests include programming language design and software engineering.