On Migrating Threads

Bernd Mathiske Florian Matthes Joachim W. Schmidt

Fachbereich Informatik
Universität Hamburg
Vogt-Kölln Straße 30
D-22527 Hamburg, Germany
{mathiske,matthes,J_Schmidt}@informatik.uni-hamburg.de

Abstract

Based on the notion of persistent threads in Tycoon[1] [MS94], we investigate thread migration as a programming construct for building activity-oriented distributed applications. We first show how a straight-forward extension of a higher-order persistent language can be used to define activities that span multiple (semi-) autonomous nodes in heterogeneous networks. In particular, we discuss the intricate binding issues that arise in systems where threads are first-class language citizens that may access local and remote, mobile and immobile resources.

We also describe how our system model can be understood as a promising generalization of the more static architecture of first-order and higher-order distributed object systems. Finally, we give some insight into the implementation of persistent and migrating threads and we explain how to represent bindings to ubiquitous resources present at each node visited by a migrating thread on the network to avoid excessive communication or storage costs.

1 Introduction and Rationale

Most of the work on migrating threads described in this paper has been carried out in the context of the European ESPRIT basic research project FIDE (fully integrated data environments). The rationale behind the FIDE project is to improve significantly the productivity in the process of building integrated, data-intensive applications. The overall goal of FIDE is to develop a consistent and small set of orthogonal language and system concepts to eliminate the historical mismatches and overlaps between independently developed component technologies.

While the initial contribution of FIDE has been the development of persistence and type system technology to overcome the mismatches between programming language and database technology, the work described in this paper addresses the overlaps and mismatches between distributed programming, transaction management and workflow management.

More specifically, we propose to generalize the well-understood programming concept of threads [POS90] to also cover persistent and migrating threads which can be used as primitive building blocks to implement long-term and distributed cooperative activities. In a recent paper [MS94], we describe the notion of persistent threads while this paper reports on our ongoing work to migrate such persistent threads between heterogeneous nodes in local or wide area networks.

The analogy with the concept of remote procedure calls may help to clarify the rationale behind our work. Remote procedure calls and migrating threads both demonstrate how to reduce successfully the complexity of distributed systems by a generalization of a well-established non-distributed programming concept:

- ▷ Programmers do not need to learn a new programming abstraction.
- ▷ Local software architectures and libraries can be scaled easily to distributed software architectures.
- ▷ Tools can be provided to optimize the mapping from high-level language abstractions to low-level communication mechanisms.
- ▷ The generalized programming abstraction interacts well with other programming language concepts like static typing, modularization and parameterization.

Our work on migrating threads reveals a strong synergy between concepts, language constructs and technologies which support persistence through time with their counterparts to support distribution through space. This synergy will become obvious in the rest of the paper which is organized as follows.

First, we present migrating persistent threads as a generalization of conventional thread concepts and motivate them as a valuable programming abstraction for the implementation of long-term distributed activities. We then introduce a terminology for data,

[1] Tycoon = Typed Communicating Objects in Open Environments [MS92, MMS94]

code and thread bindings (section 3) and basic thread operations as well as related typing aspects (section 4). This is the basis for a comparison of alternative models for distributed programming in section 5. These range from a client-server oriented programming style via higher-order languages and object migration approaches to migrating threads. In section 6 we discuss the difficult binding issues arising in the context of thread migration. Based on our classification of thread resources we describe how to achieve adequate binding support for standard programming situations.

2 Thread Persistence and Mobility

A thread describes a single sequential flow of control in a program. Having multiple threads in a program means that at any instant the program has multiple points of execution, one in each of its threads. Unlike operating system processes, multiple threads execute within a common address space, permitting multiple threads to access shared resources.

Today, threads are a well-established and standardized [POS90] programming abstraction suitable for short-lived and non-distributed applications. For example, loosely coupled high-level activities like text editing, spell-checking, typesetting, and screen refreshing can be mapped to separate concurrent threads that execute different code against shared data. Moreover, there is a rich repertoire of explicit and implicit synchronization mechanisms for thread coordination and communication (semaphores, mutexes, condition variables, channels, message queues, rendezvous', transactions).

However, the traditional thread concept exhibits severe limitations for cooperative work as required by "real" data-intensive applications, as it covers only one segment of the space/time matrix depicted in figure 1:

- ▷ The lifetime of a thread is limited by its enclosing operating system process.
- ▷ The location of a thread is limited by a single (persistent) address space.

As described in [MS94], a generalization towards *persistent threads* makes it possible to map also long-term activities (business processes or workflows) to threads. For example, the handling of a customer request by a clerk in a service department can be modeled by a thread that runs for several days or even weeks, thereby outlasting multiple activations of the same program. In a type-complete persistent language, persistent threads can be stored, for example, as attributes of database tables and they can be manipulated by user-defined queries.

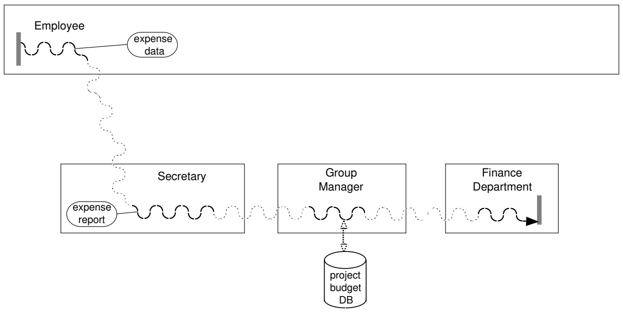

Figure 2: Implementing a distributed activity by thread migration

If one adopts an activity-oriented view of distributed applications, it is desirable to be able to express distributed workflows directly by *migrating threads* that span multiple network nodes and independent databases. For example, figure 2 and the associated Tycoon script in figure 3 describe an expense report activity in a company by means of a single thread. This thread migrates through different departments (sites) and carries around information about its past history. Being invoked by an employee, it is equipped with various bindings to mobile resources like the attributes of the expense report and immobile resources like a project budget databases maintained by the group manager. Passing through the secretary's and the group manager's site, it eventually reaches the finance department where a money transfer is initiated.

In a conventional system, activity persistence is implemented by writing the full thread state to a file or a database. To continue an activity after a process restart, its stored state has to be reloaded and a new thread has to be created in order to resume code execution at the last instruction executed in the previous session. Similar techniques have to be applied for activity migration. This "manual" approach to thread persistence and migration fails for non-trivial long-term and distributed activities due to the following technical difficulties:

- ▷ Bindings between variable names in the code (e.g., *expenseReport*, *projectBudgetDB* in figure 3) and their associated entities have to be reestablished. (Thread bindings are discussed in more detail in section 3.)
- ▷ Depending on the power of the language at hand (loops, function calls, recursion, exception handling, ...) it may be very difficult to recreate a thread that is in the correct execution state (program counter, evaluation stack, exception handler).
- ▷ It is often necessary to transmit the state and

Cooperation Mode	same time (single session)	different time (multiple sessions)
same location (single address space)	local volatile threads	local persistent threads
different location (multiple address spaces)	migrating threads	migrating persistent threads

Figure 1: Thread-based implementations of cooperative activities

```
migrate to Employee do
  repeat
    let data = getExpenseDataFromUser()
    migrate to Secretary do
      let expenseReport=compileReport(data)
    end
  until valid(expenseReport)
end
migrate to GroupManager with remote
  projectBudgetDB :ProjectBudgetDB
do
  update(projectBudgetDB, expenseReport.total)
end
migrate to FinanceDepartment do
  transferMoney(expenseReport.total)
end
```

Figure 3: Tycoon script for a migrating thread (expense report)

code information of a thread separately and to apply ad-hoc dynamic code binding mechanisms at the receiver side prior to thread restart.

In our Tycoon system and language, persistence and mobility is the default for all data, code and thread entities. Only immobile and volatile resources require special treatment by the programmer (see section 6). This relieves the programmer from low-level thread implementation details and turns threads into a suitable programming primitive for distributed, data-intensive applications.

3 Thread Bindings

In this section we introduce a terminology for the description of data, code and thread bindings (c.f. [MS94]) that we use in the rest of the paper to describe the Tycoon thread semantics, to discuss selected thread implementation aspects and to compare different models for distributed programming.

A binding is an association between a name and a computational entity from a specific semantic domain [Str67, MABD88]. We also say that a name is bound to a computational entity. An environment is a (possibly ordered) collection of bindings. Names are used to identify entities in an environment. Different names can be bound to the same entity (sharing, aliasing).

Entities can be atomic (like integers or booleans) or structured (like records, objects or functions). Structured entities typically consist of environments. For example, the fields of a record lead to bindings from field names to other entities. Therefore, bindings can be used to model (recursive) relationships between entities.

Entities can be flat (like records) or nested (like functions in Algol-like languages). In a nested entity, names bound in a global outer environment are automatically visible in a local inner environment.

The binding of names to structured entities naturally leads to the concept of a transitive closure of bindings that underlies many persistence and migration models: any entity reachable through a chain of bindings from a persistent (mobile) entity becomes persistent (mobile), too. This approach which is also adopted in Tycoon decouples the lifetime and mobility of an entity from its type (orthogonal persistence [AB87], orthogonal mobility) and should be seen in contrast to systems and languages where the programmer has to tag persistent and mobile objects explicitly at creation-time. The latter systems provide a weaker notion of referential integrity since bindings from persistent (mobile) to volatile (immobile) objects are replaced at transaction-commit (migration-time) by bindings to a distinguished NIL entity or they may even become undefined.

We distinguish three categories of structured entities:

Data describes the persistent state of an information system by a collection of computational entities related through bindings. The structure (types) of the entities and their bindings are described by types. For example, the database schema of the *projectBudgetDB* at site *GroupManager* describes the signatures of the local persistent bindings between project and account entities stored at that site.

Code is a description of operation sequences that query and update volatile or persistent entities and bindings. Code can be expressed by means of imperative or declarative, high-level or low-level programs and scripts. For example, the imperative script in figure 3 is written at a rather high level of abstraction and involves static bindings to further code entities (*getExpenseDataFromUser*, *compileReport*, ...) but also to mobile (*data*, *expenseReport*) and immobile (*projectBudgetDB*) data entities.

Threads are representations of code in the process of being executed. A thread is created by submitting a (non-parameterized) code fragment like the expense report script in figure 3 and (persistent) data to an evaluator. Multiple threads executing the same code typically have different local bindings (*expenseReport*) but shared global bindings (*projectBudgetDB*).

The semantics of the thread evaluator can be defined inductively by rules that map thread states to thread states and that perform side-effects on data [MS94]. A thread state subsumes bindings to the code fragments currently being executed and a dynamic environment that records the current bindings from names occurring in the code to local and global entities. In most imperative programming and query language implementations, thread states are represented as records that reference stacks of so-called "activation records", one for each function or query invocation.

4 Thread Operations and Typing

Following Tycoon's add-on approach to data modeling [MS91, MS92], the **migrate to** construct utilized in figure 3 is not built into the core Tycoon system. Instead of this, migrating threads are provided by a hierarchy of library abstractions and Tycoon's extensible grammar [CMA94] is exploited to provide the necessary "syntactic sugar" to hide the underlying infrastructure from high-level workflow script programmers.

Tycoon's core thread functionality is provided by a library module that exports the following exceptions, types and functions:

```
interface Thread export
  error, abortion :Exception
  T(R <:Ok) <:Ok
  State <:Ok
  runningState, blockingState, suspendedState,
  terminatedState, exceptionState :State
  state(R <:Ok  thread :T(R)) :State
  new, fork(R <:Ok  f(self :T(R)) :R) :T(R)
  suspend, run, abort, kill(R <:Ok  thread :T(R)) :Ok
  join(R <:Ok  thread :T(R)) :R
  throw(R <:Ok  thread :T(R)  exc():Ok) :Ok
  catch(R<:Ok thread:T(R) handler(exc():Ok):Ok):Ok
  atomic(R <:Ok  action() :R) :R
  ...
end
```

In Tycoon, threads are *typed* first-class values that can be stored in variables, passed as parameters, embedded into (persistent) data structures and transmitted between network sites. For example, a variable of type *thread.T(Person)* can only hold a thread that evaluates code which returns a value of type *Person*. Tycoon's polymorphic typing makes it possible to define both, generic functions that work uniformly on threads with an arbitrary result type R (*new*, *fork*, *suspend*, ...), as well as user-defined functions that depend on a specific thread result type.

Without going into details, the functions *throw* and *catch* exploit the presence of higher-order functions in Tycoon to unify asynchronous signal handling with exception handling. For example, the function call *thread.abort(otherThread)* raises the exception *thread.abortion* in *otherThread*. In addition to critical sections (*thread.atomic*), the *thread* module exports further synchronization primitives (semaphores, mutexes and condition variables) not shown here.

Building on these thread primitives, other Tycoon library modules add mechanisms to create a portable linear byte stream representation of a thread value, to establish a stream connection between network sites, to address network sites based on roles and logical identifiers, to atomically migrate running threads between network sites, etc. All these modules are loosely coupled and (by virtue of polymorphic typing and structural dynamic type checking across sites) strongly typed. This makes it possible to modify selected modules, for example, to experiment with different addressing and coordination protocols between migrating threads while ensuring a certain degree of overall system consistency.

5 Models for Distributed Activity-Oriented Systems

In this section we compare migrating threads with other programming models for distributed applications like remote procedure calls and (higher-order) distributed object management. We argue that in particular for activity-oriented tasks like workflow management it is desirable to add threads as mobile resources to these more traditional models. Our comparison is based on the uniform view of data, code and thread bindings introduced in section 3.

In the following we restrict ourselves to distribution models that scale to heterogeneous, federated and widely distributed environments. Therefore, we do not discuss concurrent or database languages based on the notion of a single (persistent) address space, like distributed databases, distributed virtual memory or distributed persistent heaps, some of which already provide multi-threading capabilities. As argued in more detail in [MMS95] we believe that the autonomy and heterogeneity requirements of larger-scale distributed systems are not addressed by these models.

5.1 Remote Procedure Call

Already in the introduction we quoted remote procedure calls as an example how to successfully generalize a well-established local programming concept to a distributed scenario. From a binding perspective, the core concept of RPCs are bindings from local to remote code entities which can be maintained for the duration of a session.

Typically, an activity is implemented by one thread of control at a single client side invoking subactivities at several server sites. A remote invocation either uses a single thread at the server side which implicitly serializes requests from multiple clients or it creates a fresh volatile thread for each incoming client request (multi-threaded server).

The plain RPC model has the crucial disadvantage for activity-oriented programming, that the code describing a long-term activity has to be fragmented, distributed and installed statically at the relevant sites. Furthermore, with conventional RPC mechanisms [Cor91, OSF93] it is very circumstantial to transmit recursive data structures.

5.2 Distributed Object Systems and Remote References

Distributed object management is viewed as a promising approach to build scalable distributed systems that are also capable of integrating legacy (database) systems by means of a unified object paradigm [MHG+92]. There are numerous proposals for specific object models like DSOM of IBM [IBM94], DOM of GTE [MHG+92], Network Objects of Modula-3 [BNOW93] and future versions of Microsoft's OLE [Mic94] and there are several related standardization efforts like CORBA of the OMG [Gro91] and the OSF DCE/DME [OSF93]. For a detailed feature analysis of these models see [NWM93, MH93].

From a binding perspective, distributed object models provide bindings to local and remote objects which aggregate data and code. Compared with the RPC model, the argument bindings for remote method invocation are generalized, since it is not only possible to transmit values but also references to other (local or remote) objects.

However, like in the RPC model there is no migration of behavior and object types have to be installed statically. As sketched in figure 4 it is necessary to split activities into separate scripts stored as object methods at different sites. Moreover, long-term and recoverable activities are difficult to implement in today's distributed object models.

Figure 4 shows how the expense report activity described in section 2 can be implemented by communicating distributed objects. Separate server objects

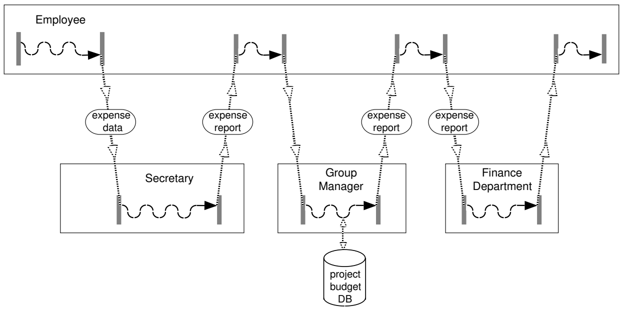

Figure 4: Implementing a distributed activity by communicating objects

describe the services available through the secretary, the project manager and the finance department. Activity control is centralized at the client (employee) site which is also responsible for exception handling at all stages of script execution.

The availability of remote references minimizes the data transfer during activity migration since each binding to a local entity at the originator site is replaced transparently on transmission by a remote reference. A remote reference identifies the originating address space and an entity local to it [JLHB88, BNOW93, Car94]. Subsequent read, write or execute access to the entity at the remote site transparently invokes a network communication with the originating site.

Conversion to remote references has clear, simple semantics since it preserves the original sharing. On the other hand it causes data fragmentation across address spaces, in particular it decreases the autonomy of migrating threads. Its usefulness depends on high site availability and low network latency.

5.3 Higher-Order Languages

Higher-order languages do not make a distinction between code and data bindings. In distributed higher-order languages like Obliq [Car94] it is therefore possible to transmit code and data uniformly between network sites. For example, by passing a function as an argument in an RPC, programmers can implement migration of behavior directly, as required by activity-oriented applications.

Moreover, since function abstraction dynamically aggregates state bindings, function migration already exhibits a limited mechanism to transmit partial evaluation states of activities across networks. For example, the Tycoon code in figure 5 utilizes four functions (*step1* through *step4*) to encode the four evaluation states of the expense report script as valid at migration time. As expected, activity migrations are implemented by passing (the closures of) the above functions to higher-order RPCs [MMS95] that sim-

```
let rec step1 = fun() begin
  let data = getExpenseDataFromUser()
  let step2 = fun() begin
    let expenseReport=compileReport(data)
    if valid(expenseReport) then
     let step3=fun(projectBudgetDB :ProjectBudgetDB)
      begin
        update(projectBudgetDB expenseReport.total)
        let step4 = fun() begin
          transferMoney(expenseReport.total)
        end
        financeDepartmentSite.execute(step4)
      end
      groupManagerSite.execute(step3)
    else
      employeeSite.execute(step1)
    end
  end
  secretarySite.execute(step2)
end
employeeSite.execute(step1)
```

Figure 5: Encoding thread states with higher-order functions

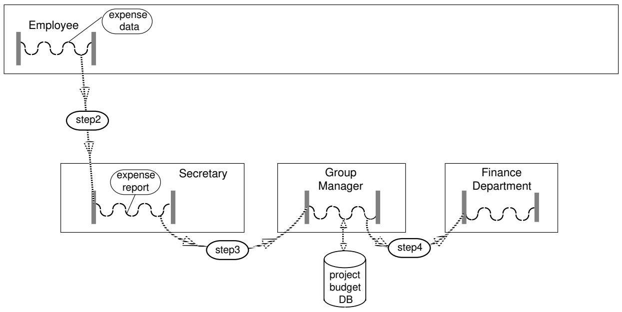

Figure 6: Emulating thread migration in a higher-order distributed language

ply execute their argument at the respective remote site.[2] In this approach iteration has to be expressed by recursion. As illustrated by figure 6, the resulting scenario is very close to our thread migration approach (see figure 2). However, additional local thread functionality is required to implement concurrent activities and activities can only be transmitted in an execution state captured by an explicit function abstraction.

5.4 Migrating Objects

While entities in many distributed object models are immobile (they never leave their creation address space), some systems also support explicit object migration (Emerald [Jul88, JLHB88], SOS [SGM89]). If an object leaves an address space, some local references are converted transparently into remote references. Conversely, if an object enters an address space, some remote references may be converted into local references.

In addition to technical advantages (load balancing, reduced communication traffic, simplified system reconfiguration) object mobility provides migration of behavior and encapsulation of state as required for activity migration.

On the other hand, the state of a long-term activity represented by a migrating object has to be encoded explicitly as a (structured) object attribute. In particular, conditional, iterative and recursive state transitions lead to complicated state encodings. This explicit encoding should be seen in contrast to persistent threads where the runtime support implicitly and efficiently maintains the state of the long-term activity.

5.5 Migrating Threads and Network Agents

If one is interested in building systems where a large number of loosely coupled activities roams the network and makes heavy use of network resources, it is desirable to avoid a cumbersome encoding of activities and to have persistent and migrating first-class threads for straight-forward activity programming.

Such a complete unification of data, code and thread binding is achieved in Emerald [Jul88, Jul89, JLHB88] and in Tycoon. Emerald makes heavy use of reference semantics for these bindings. In particular, object state attributes and thread stack frames tend to be fragmented across multiple network nodes. Therefore, the main focus of Emerald is on applications in local area networks with high site availability and low network latency. As discussed in section 6, Tycoon provides a spectrum of binding mechanisms to handle also other distribution scenarios, like autonomous network agents on global electronic marketplaces, a vision sketched in [Whi94, Way94].

Clearly, migrating threads are not a replacement for todays established distributed programming mechanisms, but they constitute a valuable programming abstraction that can be integrated smoothly into many distribution models.

6 Binding Techniques for Thread Resources

In this section we discuss the central issue in thread migration, namely how to transmit a thread state which represents a set of transitive bindings to data, code and other threads between address spaces. After a classification of thread resources we present tailored

[2] In addition *groupManagerSite.execute* passes a local binding (*projectBudgetDB*) as a parameter to *step3*.

binding mechanisms provided in Tycoon for local and remote as well as for mobile and immobile resources.

6.1 Classification of Thread Resources

In order to determine appropriate binding mechanisms for resources that are involved in the migration of a thread, we first classify these resources based on their implementation technology:

Tycoon resources are all language entities (data, code, threads) which are defined entirely in Tycoon libraries or application code and which are therefore managed by the Tycoon runtime system.

External resources are conceptually pre-existing resources that are not under direct control of the Tycoon runtime system. These resources include files, communication channels, input or output devices, graphical elements (windows, buttons, menus) or programs (mail tool, word processor). External resources are created by external library code written in C or C++ which is available to Tycoon programmers via typed language gateways. In section 6.6 we sketch a method that allows volatile external resources to "accompany" persistent and migrating threads through time and space.

In general, every atomic and structured Tycoon entity is persistent and **mobile**. If a component binding of a structured Tycoon entity (tuple, array, function, thread, ...) refers to an external resource, its persistence and mobility has to be ensured by explicit Tycoon code (see section 6.6).

Some resources like platform-dependent software components or hardware devices are inherently **immobile**. In addition it may be necessary to regard some resources (for example, a huge database or a licensed piece of software) as immobile.

A further classification is based on the scope of resource definitions:

Local resources are defined in the inner lexical scope of a thread code definition and they are allocated dynamically for each thread (e.g., *data* and *expenseReport* in figure 2).

Global resources are defined in the outer lexical scope of a thread code definition and they are potentially shared by multiple threads (e.g., the procedure definitions in figure 2).

The following important cases of distributed resources require special binding methods:

Remote resources are available at a remote site only. For example, *projectBudgetDB* in figure 2 is available exclusively at the group manager's site and requires a dynamic binding on thread migration from the secretary's site to the group manager's site.

Ubiquitous resources are available at all sites visited by a thread. Standard examples are the thread migration software itself (module *thread* in section 4), operating system functions and program libraries that serve rather general purposes like GUI and network programming. Ubiquitous resources are usually stateless but an application-specific notion of resource equivalence frequently leads to a more general interpretation of ubiquity.

6.2 Shipping Mobile Resources

We explain the shipping of mobile resources using the following **migrate to** statement taken from figure 3:

let *data* = ...
migrate to *Secretary* ***do***
  let *expenseReport* = *compileReport(data)*
end

Assuming that the **migrate to** command is executed by a thread t running at site *Employee*, the migration of t is performed as follows. First, t is suspended at site *Employee*. Then, t is shipped to site *Secretary*. Finally, t is resumed at site *Secretary*. The notation **migrate to** ... **do** is a mere syntactic sugar realized by Tycoon's extensible syntax [CMA94] for Tycoon library calls which perform these elementary thread manipulation and data communication steps.

As explained in section 3, the thread t contains a continuation binding to the code still to be executed. This code fragment (***let*** *expenseReport* = *compileReport(data)*) in turn is represented as a function closure with bindings to all its free variables (*compileReport* and *data*). The free variables in the example above constitute a local thread resource (*data*, a data entity) and a global thread resource (*compileReport*, a code entity).

The basic semantics of thread shipping in Tycoon is a *deep copy* operation between address spaces. Therefore, the thread t at site *Secretary* contains direct local bindings to all mobile resources transitively reachable from t.

User-defined bulk data structures like lists, trees, database tables, etc., and also nested entities referring to code and threads are handled correctly by the Tycoon system. Programmers have to be aware that deep copying does not preserve sharing of mutable locations and may lead to high storage and communication costs.

In order to minimize the transitive referential closure of an entity, its representation must not include (indirect) references to entities which are irrelevant for its further use. A counterexample is a linked-list representation of function closures as found, for example, in Napier88 [MBCD89] where nested functions can unattendedly capture indirect references to bulk structures, as for instance the persistent root of the object store. This virtually prevents code and thread transmission and is also a severe obstacle to garbage collections. To avoid these problems, Tycoon uses flat function closures of minimal size determined by a static binding analysis during code generation.

6.3 Working with Immobile Resources

There are two alternatives to handle immobile resources like the database *projectBudgetDB* in Fig 3.

- ▷ Thread migration to the remote site (see figure 2);
- ▷ Explicit communication with the remote site where the immobile resource is located.

In Tycoon the latter can be achieved by a remote procedure call (RPC) facility which is portable across different middleware architectures. As described in [MMS95], there are two Tycoon RPC implementations based on ONC-RPC (also known as Sun-RPC) and BSD sockets respectively. A third implementation which utilizes DCE as its communication medium is in preparation.

6.4 Dynamic Binding to Remote Resources

In a thread script, only the static bindings to local resources can be checked at compile time against the local database schema. In a loosely coupled system where remote sites can perform schema updates between thread script compilation time and thread migration time, it is necessary to perform *dynamic* bindings to remote resources (e.g. *projectBudgetDB* in figure 3). In order to locate a remote resource within the destination address space, an ubiquitous *name service* like the network object import mechanism in Modula-3 [BNOW93] or Obliq [Car94] can be used to identify resources based on a string value. This requires a dynamic type check between the actual resource type and the types used in script programming and bears the risk that a type error is detected after migration only.

As an alternative, the types of the remote resources can be attached already to the type specification of a remote *migration engine*. A migration engine is an RPC server that accepts a thread, binds it dynamically to local site resources and then resumes the thread. In Tycoon, a remote migration engine is typed based on the signatures of its resources. Therefore, mismatches between remote types and the types

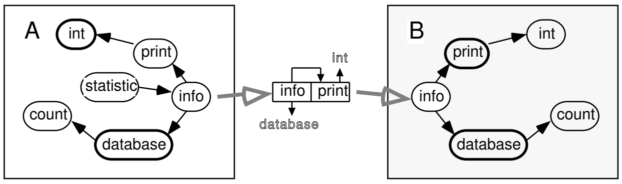

Figure 7: Rebinding ubiquitous resources by means of dynamic linking

used in thread scripts are detected already at server binding time. Thus type errors are limited to the "departure site and can only happen once per connection.

Syntactically, the name and the type of each remote resource is listed in the **with remote** clause of a migrate statement. The scope of these identifiers is restricted to the block enclosed by the keywords **do** and **end** (see also figure 3):

migrate to *GroupManager* ***with remote***
projectBudgetDB :ProjectBudgetDB
do
update(projectBudgetDB, expenseReport.total)
end

To summarize, Tycoon thread scripts are fully statically type checked at their originating site. Whenever two sites establish a network connection (that may lead to a large number of thread migrations or RPC calls), the resource type definitions of the two sites are verified to be (structurally) compatible.

6.5 Access to Ubiquitous Resources

Experience with non-trivial activity-oriented applications shows that in order to avoid excessive communication or storage costs on thread migration, system support is needed to handle bindings to ubiquitous resources present at each node visited by a migrating thread on the network.

From the programmer's viewpoint, it suffices to mark ubiquitous resources with a call to the function *register* of the module *ubiquitous*.

ubiquitous.register(windowManager)
ubiquitous.register(list)
ubiquitous.register(database)

In the example above, the Tycoon modules *windowManager*, *list* (polymorphic lists) and an application-specific module *database* are tagged as ubiquitous resources. The argument of the *register* function can be an arbitrary Tycoon entity; it is not constrained to coarse-grained module values. This is a generalization of the limited dynamic linking capabilities of other distributed programming systems (see e.g., Emerald [Jul88] and SOS [SGM89, Sha93]).

On transmission between address spaces, bindings to entities explicitly marked as ubiquitous are replaced by symbolic references. At the receiver site, these symbolic references are replaced by corresponding local bindings. In principle, a dynamic type check has to be performed during this *dynamic linking* to ensure that the type of the remote resource matches the type of the resource at the originator site. In the Tycoon implementation, this expensive repeated operation can be avoided by global identification mechanisms (for details, see [MMS95]).

The dynamic linking of ubiquitous resources is most useful for immutable values (code, literals, external bindings) replicated over the network. However, there are also some situations where a migrating thread has to bind to mutable repositories at multiple network sites. For example, a migrating thread might operate on multiple databases, file systems or ftp servers or produce side-effects on multiple screens, fax machines, ... while roaming the network.

The following Tycoon code shows the definition of the function *info* in the module *statistics* which is bound to the ubiquitous modules *database* and *print*.

module *statistic* ***import*** *database print*
export let *info() = print.int(database.count())*
end;

The left hand side of figure 7 illustrates the bindings within the address space A that contains the module *statistics*. If a thread that uses solely the function *info* is transferred to another address space B, the deep copy operation stops at the ubiquitous resources *database* and *print*.

6.6 Recreation of Volatile Resources

Volatile resources like window handles and bindings to C data structures outside of the Tycoon persistent store can be registered with the module *volatile* in order to make them pseudo-persistent. Such volatiles are recreated automatically following a system restart or a migration operation and they are destructed automatically preceding shutdown, rollback or migration operations.

The order in which volatiles are registered is significant, because typically there exist bindings from “younger” to “older” volatiles. If programmers register volatiles in creation order, the automatic recreation operations follow the original creation sequence and destruction happens in reverse order. A similar mechanism is embodied in the SOS system, where resource dependencies are defined by naming *prerequisite objects* [SGM89] which have to be recreated before their dependent objects.

Each Tycoon address space contains a global data structure referencing all volatiles under the control of the module *volatile*. The elements of this data structure never migrate; instead they are recreated automatically at the receiver site. Volatiles interact with the Tycoon garbage collector through “weak references” [HKN93].

7 Concluding Remarks

We propose to enrich today’s distributed programming models with higher-order concepts like first-class persistent and migrating threads and functions to provide better support for long-lived distributed activities like process modeling [CKO92], workflow management and network agent programming [Whi94, Way94].

At the system level, thread migration can be supported uniformly by techniques that are well-established in the persistent programming language world: stream representations of tagged object-graphs, portable code formats, function closures and polymorphic structural type checking. However, our experience with non-trivial activity-oriented applications shows that in order to avoid excessive communication or storage costs, additional system and language support has to be provided to support bindings to remote and ubiquitous resources.

More work is required to scale thread synchronization and coordination to a distributed, persistent scenario and to investigate the relationship between threads and transactions [BDS+93].

References

[AB87] M.P. Atkinson and P. Bunemann. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2), June 1987.

[BDS+93] Y. Breibart, A. Deacon, H.-J. Schek, A. Sheth, and G. Weikum. Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *ACM SIGMOD Record*, 12(3):23–30, September 1993.

[BNOW93] A. Birell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *14th ACM Symposium on Operating System Principles*, pages 217–230, June 1993.

[Car94] L. Cardelli. Obliq: A language with distributed scope. Technical report, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, June 1994.

[CKO92] B. Curtis, M.I. Kellner, and J. Over. Process modelling. *Communications of the ACM*, 35(9), September 1992.

[CMA94] L. Cardelli, F. Matthes, and M. Abadi. Extensible grammars for language specialization. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Proceedings of the Fourth International Workshop on Database Programming*

Languages, Manhatten, New York, Workshops in Computing, pages 11–31. Springer-Verlag, February 1994.

[Cor91] J.R. Corbin. *The Art of Distributed Applications.* Sun Technical Reference Library. Springer-Verlag, 1991.

[Gro91] Object Management Group. The common object request broker: Architecture and specification. Document 91.12.1, Rev. 1.1, OMG, December 1991.

[HKN93] J. Horning, P. Kalsow, J. McJones, and G. Nelson. Some useful Modula-3 interfaces. Technical report, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, December 1993.

[IBM94] IBM Corporation, Publication No. SR28-5570. *Object-Oriented Programming using SOM and DSOM*, August 1994.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions of Computer Systems*, 6(1):109–133, February 1988.

[Jul88] E. Jul. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, 1988.

[Jul89] E. Jul. Migration of light-weight processes in Emerald. *Operation Systems Technical Comittee Newsletter*, 3(1):25–30, 1989.

[MABD88] M. Morrison, P.T. Atkinson, A.L. Brown, and A. Dearle. Bindings in persistent programming languages. In *SIGPLAN Notices*, volume 23, pages 27–34, April 1988.

[MBCD89] R. Morrison, A.L. Brown, R. Connor, and A. Dearle. The Napier88 reference manual. PPRR 77-89, Universities of Glasgow and St Andrews, 1989.

[MH93] F. Manola and S. Heiler. A "RISC" object model for object system interoperation: Concepts and applications. Technical Report TR-0231-08-93-165, GTE laboratories Inc., Waltham, MA (USA), August 1993.

[MHG+92] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie. Distributed object management. *International Journal of Intelligent and Cooperative Information Systems*, 1(1), March 1992.

[Mic94] Microsoft Corporation. *Microsoft Office Developer's Kit*, 1994.

[MMS94] F. Matthes, S. Müßig, and J.W Schmidt. Persistent polymorphic programming in Tycoon: An introduction. FIDE Technical Report FIDE/94/106, Fachbereich Informatik, Universität Hamburg, Germany, August 1994.

[MMS95] B. Mathiske, F. Matthes, and J.W. Schmidt. Scaling database languages to higher-order distributed programming. March 1995. (submitted for publication).

[MS91] F. Matthes and J.W. Schmidt. Bulk types: Built-in or add-on? In *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, September 1991.

[MS92] F. Matthes and J.W. Schmidt. Definition of the Tycoon language TL – a preliminary report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

[MS94] F. Matthes and J.W. Schmidt. Persistent threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, pages 403–414, Santiago, Chile, September 1994.

[NWM93] J. Nicol, T. Wilkes, and F. Manola. Object orientation in heterogeneous distributed computing systems. *Special Issue on Heterogeneous Processing*, June 1993.

[OSF93] OSF. *OSF DCE Administration Guide – Core Components.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[POS90] Portable operating system interface for computer environments (POSIX). Federal information processing standards publication NBS-FIPS-PUB-151-1, National Bureau of Standards, 1990.

[SGM89] M. Shapiro, P. Gautron, and L. Mosseri. Persistence and migration for C++ objects. In *In Proceedings of the European Conference on Object Oriented Programming, Nottingham, GB*, July 1989.

[Sha93] M. Shapiro. Flexible bindings for fine-grain and fragmented objects in distributed systems. Rapport de Recherche 2007, INRIA, Domaine de Voluceau, Rocquencourt 78153 Le Chesnay Cedex, France, August 1993.

[Str67] C. Strachey, editor. *Fundamental concepts in programming languages.* Oxford University Press, Oxford, 1967.

[Way94] P. Wayner. Agents away. *BYTE*, pages 113–118, May 1994.

[Whi94] J.E White. Telescript technology: The foundation for the electronic marketplace. White paper, General Magic Inc., Mountain View, California, USA, 1994.