Scaling Database Languages to Higher-Order Distributed Programming

Bernd Mathiske Florian Matthes Joachim W. Schmidt
Universität Hamburg, Vogt-Kölln-Straße 30
D-22527 Hamburg, Germany

July 2, 1996

Abstract

We describe the Tycoon[1] approach to scale the successful notion of a uniform, type-safe persistent object store to communication-intensive applications and applications where long-term activities are allowed to span multiple autonomous network sites. Exploiting stream-based data, code and thread exchange primitives we present several distributed programming idioms in Tycoon. These programming patterns range from client-server communication based on polymorphic higher-order remote procedure calls to migrating autonomous agents that are bound dynamically to network resources present at individual network nodes. Following Tycoon's add-on approach, these idioms are not cast into built-in syntactic forms, but are expressed by characteristic programming patterns exploiting communication primitives encapsulated by library functions. Moreover, we present a novel form of binding support for ubiquitous resources which drastically reduces communication traffic for modular distributed applications.

1 Introduction and Motivation

Database programming languages have improved significantly the quality of data-intensive applications by contributions on two levels. At the language level, they provide flexible naming, typing and binding mechanisms between all computational entities relevant for a data-intensive application based on an integrated model for persistent (bulk) data, code and threads [AB87, MS94]. At the system level, they provide a matching integrated system technology (like tagged object representations, iteration abstractions over multiple bulk types, garbage collection, persistent abstract machines, portable code representations) that overcomes several severe mismatches of non-integrated database environments.

In this paper we argue that these two contributions to the persistence of data, code and threads over time are also highly relevant to the distribution of these entities across space, in particular, across multiple autonomous sites in heterogeneous networks. More specifically, we report on our ongoing work in scaling the Tycoon[1] system [MS93] to use Tycoon's persistent higher-order language [MS92] as a powerful scripting language for distributed applications.

We believe that it is crucial for the DBPL community to address the difficult problem how to scale the highly successful notion of a uniform, type-safe persistent object store to communication-intensive applications and applications where long-term activities are allowed to span multiple autonomous sites (possibly distributed in a world-wide network [MMS95, Whi94]).

We therefore propose to enrich the mostly reference and sharing-oriented object models of database languages with appropriate binding support for copy and replication-oriented applications that perform data, code and thread exchange via linear, portable, stream-based representations. Moreover, it is necessary to develop distributed programming idioms exploiting the higher-order concepts of modern DBPLs that help application system builders to handle replication, recovery and security issues based on application- or domain-specific knowledge.

This paper is organized as follows: After a brief Tycoon introduction (section 2) we describe stream-based data, code and thread exchange as the basis for communication between autonomous sites (section 3). Section 4 presents typical programming patterns used in distributed Tycoon applications which rely heavily on Tycoon's higher-order language

[1] Tycoon: Typed communicating objects in open environments.
This work was supported in part by ESPRIT III Basic Research Action 6309 ($FIDE_2$).

model where functions, threads and types are true first-class citizens. These programming idioms range from client-server communication based on polymorphic higher-order remote procedure calls to migrating autonomous agents that are bound dynamically to network resources present at individual network nodes. Following Tycoon's add-on approach [MS91], these idioms are not cast into built-in syntactic forms, but are expressed by characteristic programming patterns exploiting communication primitives encapsulated by library functions. Using Tycoon's extensible syntax [CMA94], it is possible for library designers to add syntax at a later stage to abstract from stereotypical programming patterns.

2 The Tycoon Language Model

The Tycoon language (TL) is an algorithmically-complete, strongly-typed, higher-order polymorphic programming language [MS92] with add-on bulk types and rich declarative query constructs. This section gives a short overview of the core TL syntax and semantics to aid the understanding of the examples in subsequent sections.

The following (recursive) type *A* denotes the type of all tuples that aggregate a floating point number named *r*, a character string variable named *s*, and a function named *foo* which takes a parameter *b* of type *A* and returns a string.

Let Rec *A* = **Tuple** *r:Real* **var** *s:String foo(b:A):String* **end**

A value *a* of type *A* can be defined through the following value binding.

let *a:A* = **tuple let** *r = 3.14* **let var** *s = "world"*
  let *foo(b:A):String = string.concat("Hello " b.s)*
end

The definition of the function *foo* uses the function *concat* exported from the *string* module in the Tycoon library to concatenate the string constant *"Hello "* with the field *s* of the tuple *b* passed as its argument. The following *recursive* value binding makes is possible to define a value *self* of type *A* that contains a function named *foo* that refers to the field *s* of its enclosing tuple.

let rec *self:A* = **tuple let** *r = 0.0* **let var** *s = "world"*
  let *foo(b:A):String = string.concat(self.s b.s)*
end

These examples illustrate virtually all binding concepts of the Tycoon language TL which have to be scaled to a distributed environment:

- ▷ Tuples aggregate static bindings and functions permit dynamic bindings through parameterization.
- ▷ Recursive bindings can be performed uniformly at the type and the value level to capture cyclic dependencies.
- ▷ TL supports both, bindings to values (*r*, *foo*, R-values) as found in functional languages and bindings to locations (*s*, L-values) as found in imperative languages [MABD90]. Locations are marked with the keyword **var** and can be updated by destructive assignments (e.g., *self.s:="new string"*).
- ▷ Functions (*foo*) and modules (*string*) are first-class TL values, i.e. they can be embedded freely into data structures, passed as parameters or returned by functions. Functions and modules may refer to free variables (*self* in the example above). The set of all free variable bindings in the static scope of a function or a module is called its *closure*.

The TL bindings above are restricted to entities (data, code, threads) that reside in a common object store which includes all volatile and persistent TL entities at a network site. It is important to note that TL entities are also allowed to contain additional bindings to *external resources* like C library functions, or file and window handles. These resources are typically non-persistent and immobile and therefore require special attention in distributed programming.

In general, a Tycoon language binding from an entity *a* to an entity *b* leads to a persistent store reference from a store object representing *a* to a store object representing *b*. Reachability-based persistence ensuring referential integrity at a network site is then enforced easily by means of a local garbage collector. For example, figure 1 shows the cyclic object store graph resulting from the binding of the recursive value *self*. The store representation of the function *foo* consists of references to the *literals* of *foo* (constant bindings to the code of *foo* and to constants values used in the source text) as well as references that constitute the *closure* of *foo* (the module *string* and the value *self*).

The signature of a polymorphic TL function is written as follows:

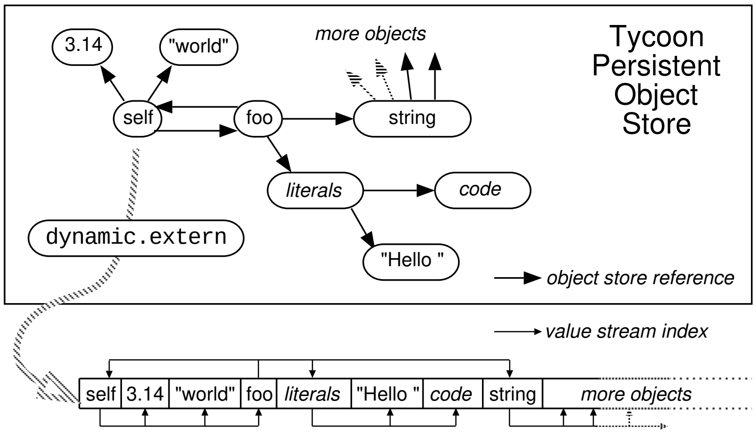

Figure 1: Linearization of a cyclic object graph

let *sort(A<:***Ok** *a:Array(A) greaterEqual(:A :A):Bool) :***Ok** = …

The function *sort* takes a type, a value and a function argument. For each type argument *A* which can be an arbitrary subtype of the top type **Ok** in the TL subtype hierarchy, the value *a* has to be an array with elements of type *A* and the function *greaterEqual* has to be a comparison function between pairs of *A* values. A discussion of TL's higher-order type system [MS92] is beyond the scope of this paper, however, it should be noted that many of the data-independent operations typical for distributed applications (shipping, aggregation, copying, …) are captured naturally by a polymorphic typing discipline.

3 Stream-based Data, Code and Thread Exchange

Similar to other modern languages like Amber [Car86], Quest [Car89], Standard ML [MTH90] and Napier88 [MBC+94], Tycoon provides a mechanism to write a deep copy of an arbitrarily complex value to a file. In Tycoon, this mechanism is generalized further since it is possible

- ▷ to linearize data, code as well as threads (partially evaluated code),
- ▷ to write to arbitrary byte streams (byte sequences in the object store, operating system files, network communication channels, …),
- ▷ to exchange byte streams between heterogeneous hardware architectures with automatic and efficient conversion of data, code and thread representations,
- ▷ to attach dynamic type information to the linear value representation to avoid type-unsafe access (this also works correctly for values of abstract data types [OTCP90]),
- ▷ to install user-defined methods which handle external (volatile or immobile) resources like windows, files or SQL tables referenced by Tycoon language values.

For example, the following TL code writes a representation of the value *self* defined in the previous section together with a dynamic type representation of its type *A* to a newly created operating system file:

let *w = writer.file("TestFile.tyc")*
dynamic.extern(w dynamic.new(:A self))
writer.close(w)

As depicted in figure 1 the full transitive closure of *self* is linearized by representing object store references through indices in the value stream. Section 5.1 describes refinements of this basic copy-based exchange mechanism to reduce significantly the amount of information that needs to be linearized (30kB in this simple example) by exploiting the semantics of certain values in the transitive closure.

The following code opens the operating system file created above for reading, reconstructs the object graph from the stream (preserving cycles and sharing), and performs a dynamic type check to ensure that the dynamically-typed value *dyn* read from the stream has type *A*.

```
let r = reader.file("TestFile.tyc")
let dyn = dynamic.intern(r)
let aCopy = dynamic.narrow(:A dyn)
reader.close(r)
```

If the dynamic type check fails, *dynamic.narrow* raises an exception that can be handled by the application program. Otherwise no further dynamic type checking is required to program with the value *aCopy*.

```
aCopy.foo(aCopy)
```

Even in a persistent language like Tycoon there are numerous applications for stream-based representations, like the exchange of compiled interfaces, modules, library descriptions between separate Tycoon object stores, the backup and logging of small databases, the switching between alternative database versions, the generation of stand-alone Tycoon boot files consisting of a main program function, and the dynamic code shipping from a WWW server to Tycoon clients.

4 Distributed Programming Idioms in Tycoon

This section presents typical programming patterns to build integrated Tycoon applications in distributed environments exploiting the binding and linearization primitives introduced in the previous sections.

4.1 Higher-Order Remote Procedure Calls

Synchronous remote procedure calls (RPCs) are the preferred technique for building client-server applications [Cor91]. Based on the flexible stream-based exchange mechanisms described in the previous section, the Tycoon library contains two generic TL modules *rpcClient* and *rpcServer* that support RPCs between separate Tycoon processes that typically run on top of separate object stores.

The following features distinguish Tycoon's RPCs from commercially standardized RPCs [Cor91, OSF93] and RPCs implemented in other persistent languages [MdS95]:

- ▷ There are no restrictions on the permissible remote function parameter types. In particular, a remote function can be *higher-order* and it can take values of abstract data types.
- ▷ It is possible to call and ship *polymorphic* functions, i.e. the type of arguments supplied to a remote function may vary dynamically from call to call.
- ▷ The binding to modules exporting remote functions can be performed fully *dynamically* at runtime; it is not necessary to invoke stub generators, to compile source texts or to relink the application code. Therefore, it is possible to program higher-order services like directory services or object brokers fully type-safe within TL itself.

Tycoon RPCs have "at most once" semantics and hide the details of the communication software which is used by the Tycoon library to implement (portable byte stream) data transmission. Currently, Tycoon supports ONC RPC (also known as Sun RPC) and BSD sockets; the use of DCE RPC is under preparation.

Communication failures are reported to TL clients by raising a dedicated exception (*rpcClient.error*) that gives access to the name, dynamic type and network address of the respective service. Application-specific exceptions raised by the remote function are propagated (including optional exception arguments) to the client site.

To illustrate the use of Tycoon's generic RPC services, assume that a module *dbOps* with interface type *DBOps* (a tuple aggretating a type, an exception value and several functions operating on an encapsulated collection of *Person* tuples) is defined in a Tycoon object store at site S and is to be made accessible via RPCs to other Tycoon sites.

```
Let DBOps = Tuple
   Let Person = Tuple name:String  age:Int  … end
   error :Exception
   insert(p :Person) :Ok
   any(predicate(:Person):Bool) :Person
   getRiscCalculation()(:Person) :Real
end
let dbOps :DBOps = tuple
   Let Person = …
   let error = exception "database.error"
   let insert(p :Person) :Ok = …
   let any = …
   let getRiscCalculation = …
end
```

In a first step, the function *new* of the module *rpcServer* is called at site S which returns a handle for an initially empty set of modules to be dispatched synchronously (i.e. by a single Tycoon thread).

let *server = rpcServer.new()*

Next, the polymorphic *register* function is used to add modules to the set of registered services at site S and to specify (optional) service names which can be used later by Tycoon clients for identification purposes.

let *registeredService = rpcServer.register(server "dbOps" dbOps)*

Note that this function depends crucially on the fact that modules (like *dbOps*) are first-class values in TL.

The simplest way to implement an RPC server is to start an infinite loop that blocks the current thread until a request is received, reconstructs the arguments from their stream-based representation, invokes the local function and returns the result (possibly an exception packet) as a linear stream to the requesting site. This loop is implemented by the *dispatch* function which is to be called at site S.

rpcServer.dispatch(server)

Tycoon provides additional functions that give a finer control over the behavior of the RPC server (polling, time-outs, conditions, remote control). Modules can be inserted and removed dynamically at a given server:

rpcServer.unregister(server registeredService)

Such a call can be made during the execution of a request, asynchronously by a concurrent thread, or after the server stopped handling requests.

At the client site C the services of the generic module *rpcClient* are used to establish an RPC connection to the *dbOps* services exported from site S.

let *remoteDBService = rpcClient.remoteService(:DBOps "dbOps" domain)*

This function performs a broadcast in a network *domain* to locate a running Tycoon RPC server that dispatches requests to a module with the name *dbOps* and a type *M* that is a subtype of the type *DBOps* specified as the first argument[2]. If no such service can be found, an exception is raised. Otherwise an abstract value of type *rpcClient.RemoteService(DBOps)* is returned that describes the remote service at site S (exact service type, service name, communication protocol, network address). Again, this example illustrates nicely the use of polymorphic typing in distributed programming since the explicit type argument *DBOps* makes it possible to completely statically type-check any further use of *remoteDBService* at the client site.

Finally, a call to the polymorphic *bind* function at site C returns a newly created local "stub" module *remoteOps* of type *DBOps*. Each function of this module ships its arguments to the RPC server at site S, invokes the corresponding function remotely, and returns the remote result as a local object store value.

let *remoteOps = rpcClient.bind(remoteDBService)*

[2] Since this matching is based on structural type equivalence, it is not important that the same type identifier *DBOps* is used at the client and server site [Nel91].

The following examples demonstrate that *remoteOps* gives fully transparent access to the (higher-order) remote functions.

let *john* = **tuple** *"John Smith" 36 …* **end**
try *remoteOps.insert(john)* **when** *remoteOps.error* **then**
  … (handle duplicate exception *)*
end
let *isAdult(p:Person):Bool = p.age >= 18*
let *a = remoteOps.any(isAdult)*   *(* pass a function as an argument *)*
let *risc = remoteOps.getRiscCalculation()* *(* return a function as a result *)*
if *risc(a) > 0.5* **then** *…* **end**

However, since there is no uniform object store that spans the local and remote site, programmers have to be aware of the implicit copy operations performed on function arguments and results.

4.2 Remote Execution Engines

In higher-order languages, remote evaluation [SG90] and *remote execution engines* [Car94] provide an interesting alternative to the traditional RPC approach described in the previous section. Instead of exporting a fixed set of remote procedures operating on encapsulated state variables, a remote execution engine exports a single, generic higher-order *execute* function that can be parameterized by arbitrary client-defined code which gains direct access to the remote state through dynamic binding.

To illustrate the idea, assume the bulk collection *persons* be defined in a Tycoon object store at site S in the static scope of an *execute* function which takes a function *f* and returns the result of applying *f* to *persons*. Since *execute* works uniformly for all result types *R*, it is defined as a polymorphic function.

let *persons :set.T(Person) = …*
let *dbEngine* = **tuple**
  let *execute(R<:***Ok** *f(:set.T(Person)):R) :R = f(persons)*
end
rpcServer.register(server "dbEngine" dbEngine)

More generally, the signature of a remote execution engine that gives unrestricted access to a value of type *Data* is defined by the following type operator:

Let *Engine(Data<:***Ok***)* = **Tuple** *execute(R<:***Ok** *f(:Data):R):R* **end**

Using the generic *rpcClient* module described in the previous section, a client can now ship arbitrary (statically-typed) queries and update operations to be performed on the remote set variable.

Let *Persons = set.T(Person)*
let *remoteService = rpcClient.remoteService(:Engine(Persons)*
                "dbEngine" domain)
let *remoteDBEngine = rpcClient.bind(remoteService)*
let *query(pers :Persons)* = **select** *p.name* **from** *p* **in** *pers* **where** *p.age>18*
let *update(pers :Persons)* = **delete** *p* **in** *pers* **where** *p.age==17*
remoteDBEngine.execute(query) remoteDBEngine.execute(update)

As illustrated by the example above, remote execution engines are particularly relevant for data-intensive applications where it is desirable to move the query (including its comparatively small closure) to the bulk data collection and not vice versa. Note that an SQL server is essentially a weakly-typed remote execution engine.

4.3 Thread Migration

The programming idiom of remote execution engines described in the previous section still adheres to the standard client-server communication paradigm. For the emerging class of workflow applications supporting multi-site business processes [HL91, Mar90] and for network agents that operate on behalf of human users in world-wide distributed networks [Whi94, Way94], this paradigm exhibits some limitations, for example:

- ▷ With today's technology it is difficult to maintain (or recover) client-server bindings for the duration of such long-term activities which may take days or weeks to complete.
- ▷ For many of these activities it is not required to return control to the originating site after completion of a subtask. Instead of this, control can be passed on directly to one (or possibly several) successor sites.
- ▷ It is difficult to split the state (context) of a complex workflow into disjoint client and server contexts. It is often more natural to incrementally accumulate a single context during evaluation that has to be carried forward from site to site.

As explained in more detail in [MMS95], we propose *migrating persistent threads* as a distributed programming idiom for such applications. For illustration purposes, we use a rather trivial workflow "get the addresses of all professors maintained by the CS department and add these to the address database of the administration".

In TL, the autonomous sites and their resources are declared as statically or dynamically bound variables using the type operator *Site* exported from the library module *agent*:

```
Let Site(Data<:Ok) = rpcClient.RemoteService(Engine(Data))
computingScience :Site(Tuple persons :Persons  ...  end)
administration :Site(Tuple addresses :Addresses  ...  end)
```

In TL, a workflow is activated by spawning an autonomous thread (*thread.fork(collectAgent)*) based on a script defined by a TL function.

```
let collectAgent(self :thread.T(Ok)) :Ok = begin
  let csDeptDB = agent.migrate(computingScience)
  let profAddr =
    select p.address from p in csDeptDB.persons where p?professor
  let adminDB = agent.migrate(administration)
  set.insert(adminDB.addresses profAddr)
end
```

The parameter *self* is a handle for the thread that executes the script (unused in this script). The script can use TL's full algorithmic power to express computations to be performed at the sites visited by the workflow. Thread migration is accomplished by calls to the function *migrate* exported from the module *agent*. It (atomically) copies the current thread to the site designated by its argument, kills the currently executing thread and resumes thread execution at the remote site, returning a binding to local resources at the remote site, as defined by the type declared for that site [MMS95].

In higher-order languages, thread migration can be emulated by adopting a continuation-passing programming style where evaluation states are encapsulated by function closures that are passed as explicit function arguments (for more details see [MMS95]). However, this approach not only leads to "cryptic" program code but also does not scale to multi-threaded agents.

As described in section 3, the shipping of a thread implies a shipping of all objects that are reachable through names in the global and currently active local scope. For example, the first *agent.migrate* call ships the code of *collectAgent*, as well as the actual objects bound to *self*, *computingScience* and *administration*. The second call additionally ships the objects bound to *profAddr* and *csDeptDB*.

In order to avoid the shipping of the full *csDeptDB* in the second migration step, it suffices to delimit the scope of this name to a nested **begin end** block and to return the result of the computation as a value from the nested block.

```
let profAddr = begin
  let csDeptDB = agent.migrate(computingScience)
  select p.address from csDeptDB.persons where p?professor
end
let adminDB = agent.migrate(administration)
```

Using Tycoon's extensible syntax [CMA94] it is straightforward to provide syntactic sugar for this particular programming idiom to make workflow scripts more readable:

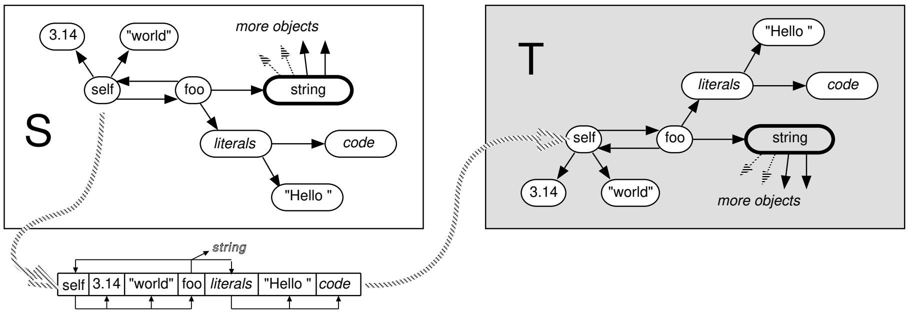

Figure 2: Dynamic rebinding to ubiquitous resources

workflow *collectAgent* **do**
let *profAddr* = **migrate to** *computingScience* **with local** *csDeptDB* **do**
select *p.address* **from** *csDeptDB.persons* **where** *p?professor*
end
migrate to *administration* **with local** *adminDB* **do**
set.insert(adminDB.addresses profAddr)
end
end

5 Binding Support for Ubiquitous Resources

Experience with communication models that involve deep copy operations shows that it is difficult to control the size of the transitive closure of functions and modules in large modular applications (see, e.g., [Wai89]). Programmers have to be careful to introduce dynamic R-value bindings or mutable L-value bindings at the right places to keep code and data fragments self-contained.

In Tycoon, for example, applications make heavy use of library functionality which includes the windowing system, the communication software, font tables, bulk data type implementations, and the reflective Tycoon compiler itself. Many of these libraries can be regarded as *ubiquitous resources* since they are installed virtually in all Tycoon objects stores at all sites. Furthermore, these libraries are practically state-less.

The characteristics of ubiquitous resources can be exploited to reduce communication traffic and to provide automatic installation mechanisms.

5.1 Dynamic Rebinding to Ubiquitous Resources

The basic idea to reduce the volume of data that needs to be communicated between sites is to represent a binding to a ubiquitous resource during object graph linearization at a source site S by a symbolic identifier which is used to rebind the copy at a target site T to the corresponding local resource.

For example, the Tycoon standard module *string* can be registered as an ubiquitous resource both at site S and T:

relink.unsafeRegister(string "module:stdenv.string")

This reduces the size of the linear representation on exchange between site S and T for the value *a* defined in section 2 from 30kB to less than 100 Bytes (see fig. 2).

A main drawback of the *unsafeRegister* function is the fact that there is no guarantee that symbolic identifiers (like "*module:stdenv.string*") are used consistently across sites. On the other hand, it is not desirable to augment each symbolic identifier with a full structural type information for the value to be identified since type descriptions for complex modules may be as expensive to transfer as the module they describe.

A first contribution to the solution of this problem is a symbol generator function in the Tycoon libraries that returns on each call a fresh (world-wide unique) universal identifier (UID, composed of platform identification, machine identification and timestamp).

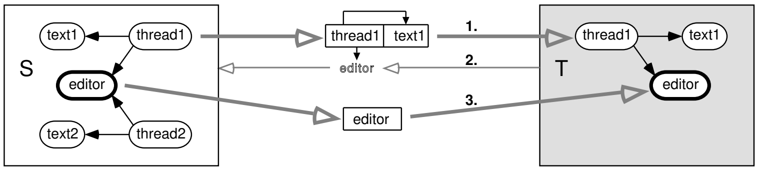

Figure 3: The three steps of automatic resource replication

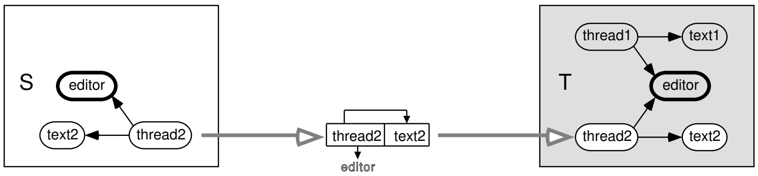

Figure 4: Dynamic relinking to an automatically replicated resource

UID values can then be embedded into complex values shipped across the network to detect whether two such values in different object stores are "semantically equivalent". UIDs are used in the Tycoon system, for example, to identify exception values and to implement the name-equivalence test required for the dynamic type check of abstract data types.

Since the compiler assigns a UID to each Tycon module, this UID can be used as a type-safe identification mechanism for module values in different stores which stem from the same compilation:

relink.registerModule(string)

For specific application domains, combinations of UID value checking and type checking can be utilized to define more elaborate mechanisms to identify "equivalent" resources.

Dynamic relinking can be applied to arbitrary TL values (modules, functions, bulk collections, …) and it blends well with all of the distributed programming idioms described in section 4.

5.2 Automatic Resource Replication

Dynamic rebinding as described in the previous section fails with an exception if an incoming symbolic identifier generated at the source site S is not registered at the target site T. If there exists a bidirectional communication channel between S and T, such a failure can be handled by requesting an explicit transmission of the missing resource from S to T which is then registered dynamically.

Figure 3 illustrates such an *automatic resource replication* in a workflow-oriented scenario where there are two Tycoon threads at site S using a shared *editor* to edit text documents that are local to the threads. Let *editor* be registered for dynamic binding. The migration of *thread1* involves shipping of the thread code, *text1* and a symbolic identifier generated for *editor* (step 1 in fig. 3). Since *editor* is not yet registered at T, the shipping of *editor* is requested explicitly (step 2), and then *editor* is shipped including its closure (step 3).

A subsequent migration of *thread2* from S to T is handled by dynamic rebinding as described in the previous section (see figure 4).

Again, automatic resource replication is orthogonal to the distributed programming idioms described in section 4 and simplifies greatly the management of distributed modular applications.

6 Related Work

Over the years there have been several studies to add distribution to persistent programming languages (see [DRV91] for some earlier references).

While the early work on DPS-algol [Wai89] attempts to give the programmer the illusion of a non-distributed persistent object store, later work in the context of Napier88 [DRV91] is based on explicit copy operations between sites and introduces a concept similar to remote execution engines as described in section 4.2. However, the code to be executed at a remote site has to perform explicit dynamic environment lookup operations to bind to resources only available at the remote site. This should be seen in contrast to our approach where dynamic type checking is performed only once, namely when a connection to a remote engine is established. To our knowledge, the concepts described in [DRV91] have not been pursued any further in the Napier project.

Munro [Mun93] describes a store-to-store communication interface at a level of abstraction similar to Tycoon's stream-based data exchange (see section 3). Munro argues that this mechanism combined with a two-phase commit protocol provides a foundation for higher-level programming abstractions.

The type-safe RPCs described in [MdS95] are generated dynamically using reflective techniques in Napier88, but they are not data type complete. For example, functions, threads or recursive values are not supported.

The Octopus (Object Closure Transplantable to Other Persistent User Spaces) described in [FD94] is a reflective language mechanism developed in Napier88 that can be used to isolate portions of closures and copy them between persistent object stores. Partial closures can be rewired, possibly in a different context, using the meta level interface supplied by Octopus. This model is more general than our binding support for ubiquitous resources, however, it is unclear whether it is suitable for the efficient handling of large-scale module libraries.

Network objects as found in Modula-3 [BNOW93], Obliq [Car94], Emerald [JLHB88, Jul88] and SOS [SGM89, Sha93] are a particularly attractive programming paradigm for distributed applications. In this model ("transparent object invocation" or "distributed objects"), a (non-persistent) object store may contain network references to objects in a remote store. All of these systems provide transparent remote method invocation, but differ substantially in other distribution aspects (object migration, method delegation, object fragmentation, higher-order functions, ...).

We plan to investigate further the implications of adding network objects in the spirit of Emerald and Obliq to versions of the Tycoon system. The transparent conversion between local and remote object references clearly simplifies the scaling of applications into a distributed environment. However, this is achieved at the expense of a drastic reduction in site autonomy and an increase in system complexity (distributed garbage collection, recovery, backup, ...).

7 Concluding Remarks

All features of the Tycoon distributed programming facilities described in this paper have been implemented without any change or enhancement of the generic language core of TL. By lifting C functions to TL almost all of the system programming could be done in a *strongly-typed*, generic language environment. The *generic* client and server stub modules contain only a few type-unsafe operations which avoid the need for reflective stub compilation.

Our initial experience using the distributed programming idioms presented in this paper for communication between Tycoon systems on Unix (Sun, IBM), Macintosh and PC hardware platforms indicates that TL compares well with other script languages for distributed programming due to the strictness and richness of its language model (polymorphic typing, exception handling, bulk type support).

Security, recovery and synchronization aspects that arise if Tycoon applications involve multiple threads activated on behalf of different human users are not treated in this paper (see, however, [RMS95]). These issues will become crucial as soon as we will use TL for shipping data, code and threads through the Internet (as a standard add-on to WWW clients and servers).

Acknowledgements

We would like to thank Miguel Mira da Silva for his helpful comments on a previous version of this paper.

References

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

[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.

[Car86] L. Cardelli. Amber. In *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[Car89] L. Cardelli. Typeful programming. Technical Report 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, May 1989.

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

[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.

[DRV91] A. Dearle, J. Rosenberg, and F. Vaughan. A remote execution mechanism for distributed homogeneous stable stores. In *Database Programming Languages: Bulk Types and Persistent Data*. Morgan Kaufmann Publishers, September 1991.

[FD94] A. Farkas and A. Dearle. Octopus: A reflective language mechanism for object manipulation. 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 50–64. Springer-Verlag, February 1994.

[HL91] K. Hales and M. Lavery, editors. *Workflow Management Software: The Business Opportunity*. Ovum Ltd., London, 1991.

[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.

[MABD90] R. Morison, M.P. Atkinson, A.L. Brown, and A. Dearle. On the classification of binding mechanisms. *Infomation Processing Letters*, 34(2):51–55, 1990.

[Mar90] R.T. Marshak. Lotus notes: A platform for developing workgroup applications. *Office Computing Report*, 15(1):2, 1990.

[MBC+94] R. Morrison, A.L. Brown, R.C.H. Connor, Q.J. Cutts, A. Dearle, G.N.C. Kirby, and D.S. Munro. The Napier88 reference manual (release 2.0). FIDE Technical Report Series FIDE/94/104, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1994.

[MdS95] M. Mira da Silva. Automating type-safe RPC. In *In Proceedings of The Fifth International Workshop on Research Issues on Data Engineering: Distributed Object Management, Taipeh, Taiwan*, IEEE Computer Society Press, March 1995.

[MMS95] B. Mathiske, F. Matthes, and J.W. Schmidt. On migrating threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*, June 1995. (Also appeared as TR FIDE/95/136).

[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.

[MS93] F. Matthes and J.W. Schmidt. System construction in the Tycoon environment: Architectures, interfaces and gateways. In P.P. Spies, editor, *Proceedings of Euro-Arch'93 Congress*, pages 301–317. Springer-Verlag, October 1993.

[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.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[Mun93] D.S. Munro. *On the Integration of Concurrency, Distribution and Persistence*. PhD thesis, Department of Mathematical and Computational Sciences, University of St. Andrews, Scotland, 1993.

[Nel91] G. Nelson, editor. *Systems programming with Modula-3*. Series in innovative technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

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

[OTCP90] A. Ohori, I. Tabkha, R. Connor, and P. Philbrow. Persistence and type abstraction revisited. In A. Dearle, G.M. Shaw, and S.B. Zdonik, editors, *Implementing Persistent Object Bases, Principles and Practice*, pages 141–153, 1990.

[RMS95] A. Rudloff, F. Matthes, and J.W. Schmidt. Security as an add-on quality in persistent object systems. In *Second International East/West Database Workshop, Klagenfurt, Austria*, Workshops in Computing, pages 90–108. Springer-Verlag, 1995. (Also appeared as TR FIDE/95/138).

[SG90] J.W. Stamos and D.K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, 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.

[Wai89] F. Wai. Distributed PS-algol. In R. Rosenber and D. Koch, editors, *In Proceedings of the 3rd International Workshop on Persistent Object Store Systems, Newcastle, NSW*, pages 126–140. Springer-Verlag, January 1989.

[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.