Journal of Universal Computer Science, vol. 9, no. 8 (2003), 730-744
submitted: 24/2/03, accepted: 30/5/03, appeared: 28/8/03 © J.UCS

LuaTS — A Reactive Event-Driven Tuple Space

Marcus Amorim Leal
(PUC-Rio, Brazil,
mleal@inf.puc-rio.br)

Noemi Rodriguez
(PUC-Rio, Brazil,
noemi@inf.puc-rio.br)

Roberto Ierusalimschy
(PUC-Rio, Brazil,
roberto@inf.puc-rio.br)

Abstract: With the goal of assessing the use of the tuple space model in the context of event-driven applications, we developed a reactive tuple space in the Lua programming language. This system, which we called LuaTS, extends the original Linda model with a more powerful associative mechanism for retrieving tuples, supports code mobility and includes a reactive layer through which the programmer can modify the behavior of the basic system calls. In this paper we describe the implementation of LuaTS and illustrate its main features with a few examples.

Key Words: distributed systems, tuple spaces, event-oriented programming
Category: D.1.3, C.2.4

1 Introduction

In spite of its widespread use in the development of distributed systems, many implementations of the tuple space model, including the original Linda model [Gelernter, 1985, Carriero and Gelernter, 1989], are not well suited for wide area network based applications. The main shortcoming of these implementations is the synchronous behavior of the calls provided to access the tuple space, which may lead to unacceptable levels of latency and failure.

Event-driven programming is gaining importance, among other reasons, because it overcomes the limitations associated with the synchronous nature of the client-server model. The most popular tuple space models in use today, IBM TSpaces [Wyckoff et al., 1998] and Sun Java Spaces [Freeman et al., 1999], support the concept of events.

In order to assess the use of the tuple space model in the context of event-driven applications, we developed a reactive tuple space that provides only asynchronous calls. This system, which we called LuaTS, was implemented in the Lua programming language [Ierusalimschy et al., 1996] using the ALua library [Ururahy and Rodriguez, 1999, Ururahy et al., 2002, Pfeifer et al., 2002].

LuaTS extends the original Linda model with a more powerful associative mechanism for retrieving tuples, supports code mobility, and includes a reactive layer through which the programmer can adapt the behavior of the basic system calls.

Reactive tuple spaces allow greater flexibility in the specification of software-component interaction, enhancing the time and space decoupling promoted by the tuple space model. Recent studies on reactive tuple spaces have focused on mobile agent coordination, an area that demands flexible and powerful mechanisms for coordinating and integrating heterogeneous components [Cabri et al., 1998, Cabri et al., 2000b, Denti et al., 1997, Denti and Omicini, 1999, Omicini and Zambonelli, 1998, Omicini and Denti, 2001, Silva and Lucena, 2001].

The remainder of this paper is organized as follows. Section 2 discusses the event-driven paradigm. Section 3 briefly introduces the ALua library. Section 4 describes the implementation of LuaTS. Section 5 presents a few examples and finally, in Section 6, we draw our conclusions.

2 Event-Driven Programming

Many systems can be best modeled as a stream of events and a set of reactions triggered by those events. In modern user interfaces, for example, a number of small graphical devices (widgets) are displayed to mediate human-computer interaction. By acting upon such widgets the user generates events that cause certain application routines to be executed. Most servers follow a similar event-driven dynamics. Event-driven programming tools usually provide conceptual abstractions that simplify the design and implementation of event-driven applications.

However, an event-driven architecture can be considered even in systems that do not have a reactive nature. Several authors suggest the use of event-driven programming in the development of wide-area network-based applications or as an alternative to multi-threaded programming [Ousterhout, 1996, Carzaniga et al., 1998]. Notwithstanding its widespread use, multi-threaded programming introduces problems that can have a significant negative impact on the development and performance of applications. For instance, applications that share resources require some kind of synchronization mechanism to control the access to these resources, introducing additional design complexity. The use of locks and other synchronization mechanisms may lead to deadlocks. Moreover, the debugging process of a multi-threaded application can be quite difficult due to the almost random way in which threads are scheduled. Finally, fine-grained synchronization and an increasing number of threads require extra system resources and frequent context switching, degrading general performance.

An event-driven system is composed by an *event dispatcher* (or event loop), an *event queue*, and *event handlers* (the piece of code that represents the reaction). Events are captured and queued until retrieved by the event dispatcher, which activates the appropriate event handler.

There are basically two ways to execute reactions. In the preemptive model each event has a priority, and the execution of a reaction will be suspended as soon as a higher priority event arrives at the event queue. In the non-preemptive model reactions are always executed until completion. Both models allow concurrency, so that reactions can be executed concurrently by multiple processes or multiple threads. Clearly this second option introduces the same sort of problems of traditional multi-threaded programming.

3 The ALua Library

ALua [Ururahy and Rodriguez, 1999, Ururahy et al., 2002] is an event-driven communication library based on the interpreted language Lua [Ierusalimschy et al., 1996]. An ALua application is composed by several processes (called agents) which can run in one or more different hosts, and communicate only through an asynchronous primitive (*send*). Each agent has a Lua interpreter and an event loop that manages user-interface and network events.

An ALua agent executes code only as a reaction to an event, such as the reception of a message. All messages exchanged between agents are composed of a chunk of code that represents the reaction to be executed by the addressed agent. Each agent runs only one thread, and always executes each reaction until completion.

ALua 2.0 [Pfeifer et al., 2002], which we used to implement LuaTS, introduced a few major changes in the original ALua system. Its most important new feature is the support of traditional communication channels (TCP/IP and UDP), which allow agents to exchange raw data and also to communicate with non-ALua applications.

4 LuaTS

LuaTS is a reactive event-oriented tuple space developed for the Lua programming environment. The system architecture (figure 1) is composed by several instances of two main modules, a client and a server. Communication is always between a single client and a single server.

The server module is composed by three layers:

Kernel — implements the main data structures and the mechanisms responsible for keeping tuples, *active calls*, and *reactions*.

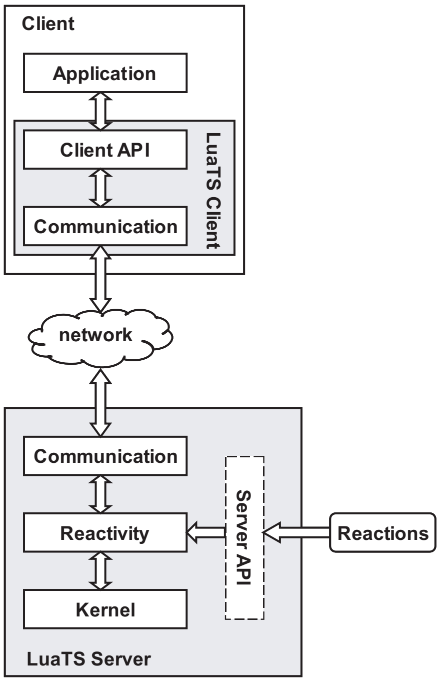

Figure 1: LuaTS Architecture

Reactive Layer — works as a filter between the kernel and the communication layer, generating side effects and eventually changing the behavior of the basic system calls according to scripts associated with triplets in the form (`call`, `client`, `template`). The reactions can be specified dynamically through a dedicated API (Server API).

Server Communication Layer — interacts with the client module. It waits for requests and, if necessary, opens new connections to send the corresponding results.

The client module acts as the tuple-space front end and is composed of two layers:

Client API — implements the calls that allow generic applications to access the tuple space.

Client Communication Layer — interacts with a LuaTS server and activates callbacks associated with requests.

4.1 The Client API

As in most tuple-space implementations, LuaTS's API is very small and simple. Due to its event-driven nature, all its operations are asynchronous. The main calls provided by this API are:

write(tuple) — inserts a tuple in the tuple space.

take (template, callback) - retrieves and removes a tuple associated with the template. The callback function will be called within the client context as soon as the request is fulfilled, receiving the retrieved tuple as its argument.

read (template, callback) — retrieves a tuple associated with the template, but does not remove it from the tuple space. The callback function will be called within the client context as soon as the request is fulfilled, receiving the retrieved tuple as its argument.

readAll (template, callback) — retrieves all the tuples associated with the template, without removing them from the tuple space. The callback function will be called within the client context as soon as the request is fulfilled, receiving a list with the retrieved tuples as its argument.

The fulfillment of any request, except `write`, is always indicated by the execution of a callback within the client context. If the server cannot immediately fulfill a `read` or `take` request, because there are no compatible tuples in the tuple space, it stores the request and fulfills it as soon as a compatible tuple is inserted. These stored requests are called *active calls*.

The `readAll` request is never stored for late fulfillment (and therefore never becomes an active call). If the server cannot fulfill it immediately, then it returns an empty table to the client, activating the registered callback.

Tuples and templates are created using two constructors:

tuple($tag, arg_1, ..., arg_n$) — this is the basic tuple and template constructor. It creates a tuple or template with $tag, arg_1, ..., arg_n$ as its fields. tag is always a string.

searchFunction ($tag, function$) - this is a special template constructor. It creates a template containing a search function and having the string tag as its first field. Search functions will be explained in the next section.

Finally, it is possible to define a time limit for storage of any particular tuple or active call in the tuple space. This limit is set with the `setTimeout` call, which takes as argument a tuple (or template) and the respective timeout in seconds.

4.2 Tuples e Templates

Tuples are modeled as ordered Lua tables containing $n+1$ fields ($n > 1$). The first field of any tuple, called *tag*, is used as an index in the storage and retrieval process, and thus must be a non-empty string. A special field with index n is used to indicate the total number of fields. A tuple field can hold any serializable object, which in Lua comprises strings, numbers, tables and the nil value. Although functions are non-serializable Lua types, tuple fields can contain strings with function code (in Lua it is possible to define a new function using this string at runtime).

Tuples are searched and retrieved through an association process that employs objects called templates. A template is modeled exactly like tuples, but it may contain, in addition to ordinary objects, a special value that represents a wildcard. If a template matches a particular tuple in the tuple space, the search is considered successful.

In LuaTS a template can also hold a special function called a search function, that is invoked by the server during the tuple retrieval process. The search function receives a tuple as its argument and tests whether the tuple matches a specific structure. If it does, the search is considered successful.

As an illustration of this process, consider an example where we want to retrieve a tuple that have two fields that bear a specific relation. The following code accomplishes this goal with the use of a search function:

```
f = [[function(t)
        -- test if fields are numbers
        if (type(t[2]) == "number") and (type(t[3]) == "number") then
          -- test the specific relation
          return (t[2] == 2*t[3])
        end
      end]]
t = ts.searchFunction("key", f)   -- create the template
ts.read(t, callback)
```

Search functions enhance the expressive power of templates and optimize the retrieval process, eventually reducing the number of calls necessary to satisfy more complex specifications and allowing searches that are impossible with traditional templates (as shown in the example above).

In ordinary tuple space implementations, tuple elements are typed objects and can be matched against special wildcard values that represent any object of a particular type. Although we could provide this feature in LuaTS, we chose not to, due to the dynamic nature of Lua and its common programming practice. In spite of that, it is easy to replicate this search semantics using the Lua function `type`.

4.3 The Kernel

The server kernel is composed of two main Lua tables, one responsible for storing tuples and active calls and another responsible for storing reactions. The latter table will be described in section 4.4. The former is a hash table indexed by the tag field of both tuples and templates. In each bucket there are two double linked lists with nodes that store tuples and active calls (figure 2).

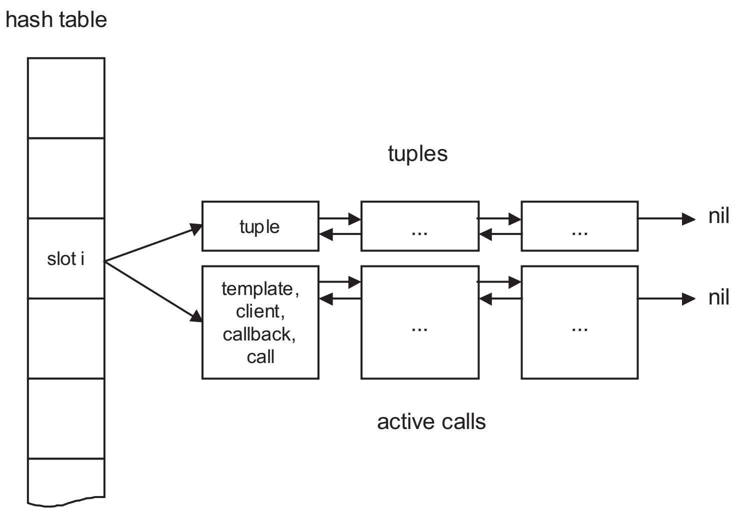

Figure 2: Kernel: main table

The tuple list stores only tuples. Each node in the active call list stores a template, a client address, the id of the callback that will be executed within the client context indicating the request fulfillment, and the original call (`read` or `take`) that was not fulfilled.

The association between tuples and templates is executed on a field by field basis. A template matches a tuple only if all its non-null fields match the tuple's respective fields and both objects have the same number of fields. Null template fields are therefore considered wildcards, and match any value in the corresponding tuple field.

When the LuaTS server receives a request with a template, it looks for a search function definition. If it finds one, it pre-compiles it and assigns the resulting function to a variable. During the search process this function will be repeatedly invoked until a valid association occurs.

A `write` request is implemented in the kernel as a very simple routine:

1. Access the bucket that has the index *tag*.
2. Access the active call list and walk through its nodes searching for a template that matches the inserted tuple.
3. If there is a match, send a copy of the tuple to the client that posted the corresponding request. If the active call is a `read`, continue to walk through the list looking for matches; otherwise finish processing.
4. If the end of the active call list is reached, then insert the tuple in the bucket's tuple list.

`read` and `take` perform a similar routine. First the tuple list is checked for a possible match. If there is a match, a copy of the tuple is sent to the client (the tuple is removed if, and only if, the request is a `take`). If there are no matches, then the request is stored in the active call list. In order to maximize the number of active calls served, `read` calls are always inserted in the front of the list, while `take` calls are inserted in the end. With this discipline when a tuple is inserted, before being tested against any `take`, it is tested against all the active `read` calls in a particular bucket.

4.4 The Communication Layers

The communication layers are implemented using a set of ALua functions that allow the management of traditional TCP/IP sockets. An application may associate event handlers with a socket. When the socket state changes (e.g., a message arrives or the connection is closed) the corresponding handler is executed.

In LuaTS each client has two independent connections with the server: one for sending requests and another for receiving the results. The client requests are buffered until the connection with the server becomes writable. After the messages are sent, the connection is kept open, basically due to optimization reasons, but the event handler associated with the "connection ready to transmit" event is switched to null. When a new request is made, the event handler is switched back to its original value.

Finally, when the client receives a result message, the socket event handler retrieves the callback registered to handle that specific request and calls it with the tuple (or list of tuples in the case of `readAll`) as its argument.

4.5 The Reactivity Layer

In the current version of LuaTS, mainly due to the lack of a robust security infra-structure, we allow the registration of reactions through the server interface only. Reactions are registered using a dedicated API call (`regReaction`) and are represented by a meta-tuple in the form `(call, client, template, reaction, delay)`. Each argument has the following meaning:

call — is the kind of call that will trigger the reaction (one of `write`, `take`, `read`, `readAll`);

client — is a table with the IP and port numbers of the client that made the call;

template — is a template to match the tuple inserted or retrieved by the call;

reaction — is a string with the code representing the reaction;

delay — is a flag that indicates when the reaction should be executed (we will explain this later on).

The client and template fields may contain a null value, which play the role of a wildcard. The other arguments must be non-null values.

Reactions are retrieved according to a routine similar to those used for retrieving tuples and active calls. A reaction meta-tuple is stored in the reaction table, indexed by the call. For each call, we use another table that is indexed by the template tag field. Calls with null templates are assigned to a special entry.

The reaction layer searches for reactions immediately before a request is executed by the kernel, and immediately after a result is produced. The delay field in the reaction meta-tuple allows the programmer to specify in which of those moments a particular reaction should be executed (if delay is a non-null value, the reaction will be executed after a result is produced).

More than one reaction may be associated with each request. In this case all reactions will be executed in sequence. However the side effects generated by one reaction cannot directly trigger a new reaction, avoiding chain reactions and minimizing the risk of cycles.

It is important to discuss how much flexibility a reactive tuple space layer should provide. Early reactive tuple space implementations, like Law-Governed Linda [Minsky and Leichter, 1995], introduced the reaction mechanism as a way to overcome security and performance shortcomings of the original Linda model. Their semantics remained similar to that of traditional models. However, more recent implementations [Cabri et al., 1998, Cabri et al., 2000b, Denti et al., 1997, Denti and Omicini, 1999, Omicini and Zambonelli, 1998, Omicini and Denti, 2001, Silva and Lucena, 2001] support almost unlimited reactions' side effects, enabling applications to completely redefine the tuple space semantics in very unorthodox ways. [Omicini and Zambonelli, 1998] for example discuss an application in which a reaction to a `read` request executes a database query that has no relation whatsoever with the tuple space, and encapsulates the result in a dynamically created tuple that is never stored.

Although the tuple space API is very simple and powerful, the tuple space model is more than just an attractive API. Of course this API can be implemented with different semantics. In some cases, we get a tuple space; in others what we really have is a framework for implementing applications that use "tuple space like" APIs.

In LuaTS, despite any possible reaction side effect, we enforce a basic semantics that cannot be changed. A `write` will always insert a tuple in the tuple space. A reaction may even modify the content of the original tuple, but nonetheless a tuple will be inserted by that request. And to modify a tuple, the programmer has to specify this explicitly. There are no shortcuts or implicit ways to change the basic tuple-space semantics.

5 Examples

In this section we illustrate the use of LuaTS with a few simple examples. Note that an event-driven tuple space can simplify the solutions to many classic problems of concurrent and distributed programming.

5.1 Marketplace

Tuple spaces have been commonly employed in the development of electronic auctions and other e-commerce applications [Freeman et al., 1999, Cabri et al., 2000a]. An online classifieds service is an interesting case that could benefit from the search-function mechanism of LuaTS. In this example we use a tuple to represent each ad. The tuples have a standard structure with information about the offered product or service. For instance,

```
tuple("auto","id=2345","Ford Focus",2001,"blue",
{"air-conditioning","CD-player"},14000)
```

describes a sale offer of a Ford Focus, model 2001, blue color, with air conditioning, a CD player, and a sale price of $14.000.

Suppose we are interested in a Ford Focus and would like to retrieve some offers. We are willing to pay $15.000 for a 2001 model, or $12.500 for a 2000 model. To implement this query we can use the following code:

```
function printOffers (t)
  if getn(t) == 0 then
    print("No sale offer found!!!")
    return
  end
  for i,v in t do
    print("Offer "..i..": "..ts.tostring(v))
  end
end


search = [[function (t)
```

```
            if t[3] == "Ford Focus" and
              ((t[4] == 2000 and t[7] <= 12500) or
               (t[4] == 2001 and t[7] <= 1500)) then
              return 1
            end
        end]]

ts.readAll(ts.searchFunction("auto",search),"printOffers")
```

The search function defines the query criteria and is passed to a template through the `searchFunction` constructor. The `printOffers` function is registered as a callback. It will be called as soon as the `readAll` request is fulfilled, printing all offers retrieved.

5.2 Readers and Writers

Controlling exclusive access to shared resources can be easily accomplished using a tuple to represent the access right. Any process interested in a resource has to acquire the respective tuple with a `take` call. To free the resource, the process simply inserts back the tuple with a `write` call.

The problem of "readers and writers" is more complex. In this problem any writer process needs mutually exclusive access to a resource, but reader processes, as a group, can access the resource concurrently. We can implement a solution to this problem using shared locks, represented by tuples with the following format:

```
ts.tuple("lock ID", numberOfWriters, numberOfReaders)
```

Readers try to acquire a tuple using the template:

```
ts.tuple("lock ID", 0, nil)
```

While writers try to acquire the tuple using the template:

```
ts.tuple("lock ID", 0, 0)
```

When the tuple is acquired, access right is immediately granted. Each process is responsible for incrementing and decrementing the corresponding `numberOfWriters` or `numberOfReaders` fields.

In the solution above, any writer will be indefinitely blocked while there is one or more readers interested in the resource. A fairer solution extends the former tuples with just one extra field:

```
ts.tuple("lock ID", numberOfWriters, numberOfReaders, delayedWriter)
```

where the `delayedWriter` field is a binary flag.

Now all readers and writers try to acquire a tuple with the template:

```
ts.tuple("lock ID", 0, nil, 0)
```

When a reader acquires the tuple, it gets immediate access to the resource. A writer, on the other hand, faces to possibilities:

- `numberofReaders` is zero — in this case the access is granted immediately.
- `numberofReaders` is not zero — in this case only a priority is granted, not the access. The access right will be acquired only when all previous readers release the resource. The writer has to set the `delayedWriter` field and wait for the tuple

  ```
  ts.tuple("lock ID", 0, 0, 1)
  ```

5.3 Reactivity

An interesting application of the reactivity mechanism is tuple-space access control. For several reasons a system administrator may want to restrict user access rights. Some users may not be allowed to remove tuples, for example. This kind of control can be implemented as illustrated by the code below:

```
function log (client, tuple, callback)
  if notAuthorized(client) then   -- checks if the client has
                                  -- rights to execute a take
    local file = appendto("log.txt")
    if file then
      write(file, format("At %s client %s:%s tried to remove %s \n",
                  date(), client.ip, client.port,
                  ts.tostring(tuple)))
      closefile(file)
    end
    ts.write(tuple)   -- reinserts the tuple in the tuple space
  end
end


ts.regReaction(ts.take,nil,nil,log,1)   -- registers the reaction
```

In this example all non-authorized attempts to remove tuples are recorded in a log file. A reaction is executed immediately after the kernel extracts a tuple requested by a non-authorized user. The extraction attempt is recorded in a log file and the tuple is reinserted in the tuple space. Notice that, in spite of the reaction, the original `take` call is executed normally, i.e. the requested tuple is removed (even though it is reinserted later) and sent to the client.

6 Conclusion

Our programming experience with LuaTS shows that the uncoupling promoted by the tuple space model added to an event-driven dynamics facilitate process synchronization and yield a much simpler execution thread. The programming task becomes easier when compared to traditional client/server and multithreaded architectures and is less error-prone.

LuaTS follows an explicit event-driven dynamics and supports asynchronous calls only. JavaSpaces and TSpaces, on the other hand, provide mainly synchronous calls. As already mentioned, they also support the concept of an event, but just through a notification service that has a fairly complex semantics. Simple tasks, such as removing a tuple that triggered an event, are not easily implemented. The programmer has to explicitly handle thread synchronization and worry about deadlocks, exactly the kind of problems that we wish to avoid with event-driven programming.

Another interesting aspect of LuaTS is its search semantics. Although a few implementations support more flexible mechanisms than the traditional template association process, as far as we know, none of those reach an expressiveness similar to that of search functions. This facility allows queries that are not possible with traditional templates, and can improve the retrieval process by reducing the number of requests necessary to satisfy complex specifications.

Many of LuaTS's capabilities depend on its code mobility support, something directly inherited from ALua and the Lua language itself. Code mobility can be informally defined as the capability to reconfigure, during runtime, the bindings between the software components of the applications and their physical location within a computer network [Carzaniga et al., 1997]. Code mobility support is generally classified as strong or weak [Fuggetta et al., 1998]. Strong mobility support the migration of code and its execution environment, i.e. its stack, global variables, registers, etc. Weak mobility, on the other hand, support only code migration. We could also define a third class of mobility support called semi-strong, that implicitly support code migration and program data, like global variables and object attributes. Java for example does not support strong mobility, but its serialization mechanism is much more powerful than those found in languages with only weak mobility support, and in our opinion belong to a different class. ALua, and therefore LuaTS, support only weak mobility. Nevertheless both systems provide functionalities that allow the programmer to define protocols for transferring methods and attributes, achieving results similar to Java serialization.

Finally, an issue that deserves special attention in future versions of LuaTS is security. The code chunks that are exchanged between hosts are not "controlled" and could be tampered with little effort. Search functions should be handled in a sandbox that supports only a subset of the Lua language, blocking access to the server's data structures and restricting I/O. Moreover, a message signature infra-structure would allow code authentication, which could enable dynamic installation of reactions by authorized clients.

References

[Cabri et al., 1998] Cabri, G., Leonardi, L., and Zambonelli, F. (1998). Reactive tuple spaces for mobile agent coordination. In Rothermel, K. and Hohl, F., editors, *Proceedings of the 2nd International Workshop on Mobile Agents*, volume 1477, pages 237–248. Springer-Verlag: Heidelberg, Germany.

[Cabri et al., 2000a] Cabri, G., Leonardi, L., and Zambonelli, F. (2000a). Auction-based agent negotiation via programmable tuple spaces. In *4th International Workshop on Cooperative Information Agents (CIA 2000)*.

[Cabri et al., 2000b] Cabri, G., Leonardi, L., and Zambonelli, F. (2000b). MARS: A Programmable Coordination Architecture for Mobile Agents. *IEEE Internet Computing*, 4(4):26–35.

[Carriero and Gelernter, 1989] Carriero, N. and Gelernter, D. (1989). Linda in context. *Communications of the ACM*, 32(4):444–458.

[Carzaniga et al., 1998] Carzaniga, A., Nitto, E. D., Rosenblum, D. S., and Wolf, A. L. (1998). Issues in supporting event-based architectural styles. In *Proceedings Third International Software Architecture Workshop*, pages 17–20, Orlando, Florida. IEEE.

[Carzaniga et al., 1997] Carzaniga, A., Picco, G. P., and Vigna, G. (1997). Designing distributed applications with a mobile code paradigm. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, USA.

[Denti et al., 1997] Denti, E., Natali, A., and Omicini, A. (1997). Programmable coordination medium. In Garlan, D. and Métayer, D. L., editors, *Proceedings of COORDINATION'97 (Coordination Languages and Models*, volume 1282 of *LNCS*, pages 274–288. Springer-Verlag.

[Denti and Omicini, 1999] Denti, E. and Omicini, A. (1999). An architecture for tuple-based coordination of multi-agent systems. *Software Practice and Experience*, 29(12):1103–1121.

[Freeman et al., 1999] Freeman, E., Hupfer, S., and Arnold, K. (1999). *JavaSpaces(TM) Principles, Patterns, and Practice.* Addison-Wesley, 1 edition.

[Fuggetta et al., 1998] Fuggetta, A., Picco, G. P., and Vigna, G. (1998). Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361.

[Gelernter, 1985] Gelernter, D. (1985). Generative communications in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112.

[Ierusalimschy et al., 1996] Ierusalimschy, R., de Figueiredo, L. H., and Filho, W. C. (1996). Lua — an extensible extension language. *Software Practice and Experience*, 26(6):635–652.

[Minsky and Leichter, 1995] Minsky, N. H. and Leichter, J. (1995). Law-governed Linda as a coordination model. *Lecture Notes in Computer Science*, 924:125–146.

[Omicini and Denti, 2001] Omicini, A. and Denti, E. (2001). From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294.

[Omicini and Zambonelli, 1998] Omicini, A. and Zambonelli, F. (1998). Tucson: a coordination model for mobile information agents. In Schwartz, David G. AND Divitini, Monica AND Brasethvik, T., editor, *1st International Workshop on Innovative Internet Information Sytems (IIIS'98)*, pages 177–187. Department of Computer and Information Science (IDI), NTNU.

[Ousterhout, 1996] Ousterhout, J. K. (1996). Why threads are A bad idea (for most purposes). Invited Talk at the 1996 USENIX Technical Conference.

[Pfeifer et al., 2002] Pfeifer, A. L., Ururahy, C., Rodriguez, N., and Ierusalimschy, R. (2002). Event-driven programming for distributed multimedia applications. In *20o.Simpsio Brasileiro de Redes de Computadores*.

[Silva and Lucena, 2001] Silva, O. and Lucena, C. (2001). T-rex: A reflective tuple space environment for dependable mobile agent systems. In *Proceedings of the III WCSF at IEEE MWCN 2001*.

[Ururahy and Rodriguez, 1999] Ururahy, C. and Rodriguez, N. (1999). Alua: An event-driven communication mechanism for parallel and distributed programming. In *Proceedings of the 12th International Conference on Parallel and Dist. Comp. Practices*.

[Ururahy et al., 2002] Ururahy, C., Rodriguez, N., and Ierusalimschy, R. (2002). Alua: Flexibility for parallel programming. *Computer Languages*, 28(2):155–180.

[Wyckoff et al., 1998] Wyckoff, P., McLaughry, S., Lehman, T., and Ford, D. (1998). Tspaces. *IBM Systems Journal*, 37(3):454–474.