Electronic Notes in Theoretical Computer Science 54 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume54.html 11 pages

Modelling Node Connectivity in Dynamically Evolving Networks

Lorenzo Bettini Michele Loreti Rosario Pugliese

Dipartimento di Sistemi e Informatica
Università di Firenze
{bettini,loreti,pugliese}@dsi.unifi.it

Abstract

Node connectivity is a key aspect of dynamically evolving networks. We address the problem of expressing, and constraining, node connectivity at a linguistic level. We extend the language KLAIM (Kernel Language for Agent Interaction and Mobility), that already provides primitives to explicitly configure the topology of networks, with the notion of *clusters* of nodes. This feature permits explicitly grouping nodes thus expressing, and constraining, their communication ability: two nodes can interact only if they are in the same cluster. Clusters can be used to model many real situations in a natural way and to easily express a number of basic properties of node connectivity, such as, e.g., locality and distribution of nodes, efficiency, and fault tolerance. However, they do not add expressive power to the language: to some extent, the extended language can be translated into the original one.

1 Introduction

Internet provides means and technologies that permit sharing many resources and services among several computers geographically distributed in a wide-area network. This is a highly dynamic infrastructure that evolves over time. For instance, new nodes can get connected to the network or existing nodes can disconnect. Connections and disconnections can be temporary and unexpected. In certain situations a temporary connection can be established "on the fly" among terminals equipped with wireless devices, and ad-hoc paths to services can be established to enable communication among components. Moreover, the topology of connections can change and nodes can move elsewhere in the net. Indeed, some nodes can be mobile devices such as, e.g., laptops, PDAs and cellular phones.

[⋆] This work has been partly supported by MURST Projects SALADIN and TOSCA.

By exploiting Internet protocols, distributed applications can exchange data independently of the underlying architectures and operating systems. However, in the context of dynamically evolving networks, assuming that the underlying network will always be available and that there will always be permanent connections among all nodes hosting resources is too strong.

Node connectivity is a key aspect of dynamically evolving networks. We can single out three different basic forms of connectivity that a node can employ:

- *Tethered mode*: Wide-area connectivity is available, thus the node is connected to any other node in the net.
- *Disconnected mode*: The connection is discontinuous but, in the time intervals when it is available, any other node in the net can be reached.
- *Untethered mode*: There is no wide-area connectivity, but a local-area connectivity may still be available, in which case, the node is connected to any other node in the same connection area.

Of course, different nodes of a network may use different forms of connectivity. The first form of connectivity represents an idealized situation that seldom takes place in practice for all nodes of a wide-area dynamically evolving network. Here are a couple of realistic, but simplified, scenarios where users experiment with the remaining two forms of connectivity.

Disconnected mode is the usual form of connectivity for a user that does not own a network access. The user works in isolation on his own documents. Some network access points are available, thus, at some time, the user can connect to the Internet, typically via a modem. Connection is required to update user's data in the central repository or to spawn a query to some server. In the last case, mobile agents could be used to manage the query and collect the results that will be delivered to the user when he reconnects.

Untethered mode is the usual form of connectivity for a mobile user that, from time to time, enters different areas where connectivity is constrained. For instance, the user may connect to computers that are within a firewall, so that the communication among such computers is available, but connection to the rest of the net is disabled. This limitation, however, could not concern all computers; typically, there might be some of them that are allowed to access the whole net and represent the gateway to the external world.

Of course, if we consider many users that work in cooperation, we can combine the previous scenarios and obtain more complex situations where some users are always on line, while others connect now and then and must synchronize their work with the remaining ones.

Recently, new paradigms and languages for programming applications for dynamic evolving networks have been proposed that are centered around the notions of *location awareness* and of *mobile agents*. A few examples of such languages are Telescript [13], Java [1], Ambients [4] and Klaim [5].

In this paper we address the problem of expressing, and constraining, node connectivity. More specifically, we want to devise suitable linguistic abstractions for managing node connectivity. Thus, we extend the language Klaim, that already provides primitives to explicitly configure the topology of networks, with the notion of *clusters* (i.e. groups) of nodes and with appropriate primitives to manage clusters. Clusters permit explicitly grouping nodes thus expressing, and constraining, their communication ability: two nodes can interact only if they are in the same cluster. The primitives for cluster management permit dynamic cluster creation and addition/removal of nodes to/from clusters. The notion of cluster turns out to be convenient to model node connectivity in a natural way and permits easily expressing a number of basic properties of node connectivity, such as, e.g., locality and distribution of nodes, efficiency, and fault tolerance. However, clusters do not add expressive power to Klaim: indeed, to some extent, the extended language can be mapped into the original one.

The rest of the paper is organized as follows (due to lack of space, all technical details have been omitted). Section 2 informally presents the language Klaim, while Section 3 introduces the extensions for managing clusters. Section 4, by means of a simple example, illustrates how to use the new features to model node connectivity. Finally, Section 5 presents a sketch of the translation of the extended language into Klaim and the extensions to the implementation.

2 The Klaim language

Klaim (Kernel Language for Agent Interaction and Mobility, [5]) is an experimental programming language specifically designed to program distributed systems composed of several components interacting through multiple tuple spaces and mobile code. It is inspired by the coordination language Linda [8], hence it relies on the concept of *tuple space*. A tuple space is a multiset of *tuples*; these are containers of information items (called *fields*). Klaim distinguishes between *actual fields* (i.e. expressions, processes, localities, constants, identifiers) and *formal fields* (i.e. variables). Syntactically, a formal field is denoted with $!ide$, where ide is an identifier. For instance, the sequence (“foo”, “bar”, $!Price$) is a tuple with three fields. The first two fields are string values while the third one is a formal field. Similarly, (**out**(“hello”)@l, “foo”) is a tuple whose first field is a process. Tuples are anonymous and content-addressable; *pattern-matching* is used to select tuples in a tuple space:

- two tuples match if they have the same number of fields and corresponding fields do match;
- a formal field matches any value of the same type, and two actual fields match only if they are identical (but two formals never match).

For instance, tuple (“foo”, “bar”, $100+200$) matches with (“foo”, “bar”, $!Val$). After matching, the variable of a formal field gets the value of the matched field: in the previous example, after matching, Val (an integer variable) will contain the integer value 300.

In Linda there is only one global shared tuple space; Klaim extends Linda by handling multiple distributed tuple spaces. Tuple spaces are placed on *nodes* that are part of a *net*. Each node contains a single tuple space and processes in execution; a node can be accessed through its *address*. There are two kinds of addresses:

- *Sites* are the identifiers through which nodes can be uniquely identified within a net.
- *Localities* are symbolic names for nodes. A reserved locality, `self`, can be used by processes to refer to their execution node.

Sites have an absolute meaning and can be thought as IP addresses, while localities have a relative meaning depending on the node where they are interpreted and can be thought as aliases for network resources. Localities are associated to sites through *allocation environments*, represented as partial functions. Each node has its own environment that, in particular, associates `self` to the site of the node.

Klaim processes may run concurrently, both at the same node or at different nodes, and can perform the following basic operations over tuple spaces and nodes:

- **in**$(t)@l$: evaluates tuple t and looks for a matching tuple t' in the tuple space located at l. Whenever the matching tuple t' is found, it is removed from the tuple space. The corresponding values of t' are then assigned to the formal fields of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available.
- **read**$(t)@l$: differs from **in**$(t)@l$ only because the tuple t', selected by pattern-matching, is not removed from the tuple space located at l.
- **out**$(t)@l$: adds the tuple resulting from the evaluation of t to the tuple space located at l.
- **eval**$(P)@l$: spawns process P for execution at node l.
- **newloc**(l): creates a new node in the net and binds its site to l. The node can be considered a "private" node that can be accessed by the other nodes only if the creator communicates the value of variable l, which is the only means to access the fresh node.

Klaim processes can be built from basic operations by using standard operators borrowed from process algebras [9], such, e.g., *action prefixing* and *parallel composition*. *Timeouts* can also be used for avoiding that processes block due to network latency bandwidth or, when retrieving information, to absence of matching tuples.

During tuple evaluation, expressions are computed and localities are translated into sites. Evaluating a process implies substituting it with its *clo-*

sure (i.e. the process together with the environment of the node where the evaluation is taking place). The difference between operation **out**$(P)@l$ and **eval**$(P)@l$ is that **out** adds the closure of P to the tuple space located at l, while **eval** sends P, not its closure, for execution at l. Therefore, if node s_1 performs an **out** of P to node s_2, when P is executed at s_2, `self` will actually refer to s_1. This means that *static scoping* is used for binding localities. On the contrary, if s_1 spawns P at s_2 with **eval**, no closure is sent: P will refer to s_2 when using `self` and *dynamic scoping* is used for binding localities.

Due to the separation between the concrete and symbolic address of nodes, allocation environments have the role of restricting sites visibility and, thus, of partially structuring Klaim nets. However, the model underlying Klaim is *flat*, namely nodes cannot enclose other nodes and all nodes are at the same level, therefore it is not completely suitable for modelling structured nets such as, e.g., the Internet. In [3] we enriched Klaim in order to transform the underlying flat model into a *hierarchical* model. In this paper we study an orthogonal issue, namely the connectivity of nodes.

3 Clusters

In the current Klaim model, a process P, running at node s_1, that knows the site s_2 can communicate with the node s_2. From an abstract point of view, nodes s_1 and s_2 are connected. Now, if P migrates to node s_3 then it is still able to communicate with s_2. Abstractly, s_3 and s_2 get connected while s_1 and s_2 can possibly get disconnected (this mechanism is reminiscent of "link mobility" in the π-calculus [10]). In the Internet, however, the knowledge of the address of a remote host is not sufficient to communicate with it, since there might be no route to the host.

In this section we enrich the Klaim communication layer by introducing the notion of cluster, thus such aspects as mobility and disconnection/reconnection of nodes can be easily modelled. A *cluster* is a collection of nodes and represents a communication medium shared by all the nodes in the cluster. In fact, two nodes can interact only if they belong to the same cluster (the semantics of Klaim operations is modified accordingly). Clusters can be used to model locality and distribution of nodes: nodes in the same cluster are "close" to each other. However, the closeness relation does not need to be specified thus, e.g., at a high abstraction level, the whole Internet can be modelled as a single cluster of nodes. A single node can belong to several clusters; this models the fact that a node can use several communication media and, thus, can be able to tolerate faults of the connection architecture. Whenever a pair of nodes belong to various clusters, anyone of them can be used for an interaction to take place. In our current framework, the choice of which cluster to use is nondeterministic, however one could easily imagine extensions where the choice depends on efficiency considerations, and, more generally, on quality of service parameters such as, e.g., bandwidth and latency. As we will see, clusters cannot be nested (although, of course, the set of nodes of one cluster could be contained into the set of the other).

Operations over clusters modify the topology of networks and can be performed only by *superprocesses* (these are denoted by $\mathbb{P}$). Superprocesses cannot migrate and cannot be used as tuple field. They are installed at a node when the network is initially configured or when the node is dynamically created by using $\mathbf{newloc}(u, \mathbb{P})$, which creates a new node with superprocess $\mathbb{P}$. The superprocess of a node can be thought of as an abstraction of that part of a network operating system that lies at the node, while standard Klaim processes are the user programs that can call for execution of system calls. In addition to the standard Klaim operations, superprocesses can also perform operations **newc**, **add** and **rm** acting over clusters. More specifically, $\mathbf{newc}(w).\mathbb{P}$ creates a new, empty cluster that can be referred to via the variable w. $\mathbf{add}(c).\mathbb{P}$ adds the node where it is running to the cluster c, whilst, conversely, $\mathbf{rm}(c).\mathbb{P}$ removes the node from c. Hence, node mobility is modelled in terms of the ability of nodes in entering to and exiting from clusters.

As an example, let us consider the net N in Figure 3 (a). Initially, nodes s_1, s_2 and s_3 are in the same cluster c_1. Afterwards, a superprocess at s_1 creates a new (empty) cluster c_2 by means of $\mathbf{newc}(w)$ (after the operation has been executed, w is bound to c_2). When $\mathbf{add}(c_2)$ is performed at s_1, s_1 is added to c_2 (Figure 3 (b)). Now, s_1 belongs to both c_1 and c_2. Finally, when $\mathbf{rm}(c_1)$ is performed at s_1, the node is removed from c_1, hence all connections among s_1 and the nodes in c_1 are broken, while processes in s_1 are still running (Figure 3 (c)).

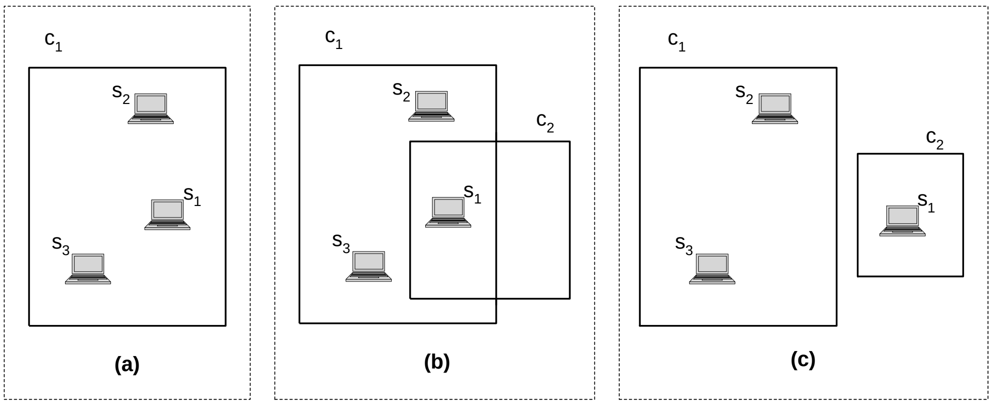

Fig. 1. Klaim with clusters: an example.

To end this section, we show how to use Klaim extended with clusters for modelling the forms of connectivity presented in the introduction.

- *Tethered Mode*: There is only one cluster and all nodes belong to it. Moreover, a node cannot be removed from the cluster, hence there are no superprocesses in the nodes. The standard Klaim model fits in this picture.
- *Disconnected Mode*: There is only one cluster and all nodes belong to it.

However, a node can be removed from the cluster and work in isolation. Hence, differently from the tethered mode, there are superprocesses running on nodes, but new clusters cannot be created.

- *Untethered Mode*: There is not a single cluster, but a set of clusters, each corresponding to a different local connection area. Superprocesses can add/remove nodes to/from clusters and can create new clusters.

4 Disconnected mode: an example

Let us consider the following scenario where disconnected mode is the form of connectivity used.

A software engineer works in isolation. Sometimes, he dials up and gets reconnected to a virtual community server in order to download updates and perform a query in the virtual community. After that, he goes off-line. Later, he reconnects to the virtual community and the results of the search will be dispatched. Moreover, he can establish a communication with one of the other community users. When he has acquired enough information he logs off. Finally, when he finishes his work, he logs on and uploads the artifacts.

Let us now consider a possible specification in Klaim that also exploits clusters and operations over them. In the net that models the scenario there is only one cluster, called *virtualc*, corresponding to the virtual community. In the cluster, there is a node for the server, called S, and one node for each user in the community. A node is on-line when it belongs to the cluster and, conversely, it is off-line when it is removed from the cluster. We just consider superprocesses and Klaim processes running at the engineer's node. We assume that, initially, the engineer's node is outside the cluster.

Connection to and disconnection from the cluster are handled by two superprocesses, *ConnectionSP* and *DisconnectionSP* respectively, that act upon receiving requests issued by means of tuples of the form (*operation*, *cluster*). Notice that disconnection has to be acknowledged so that the requiring process can be sure that the node has gone off-line, while connection does not, because any non local operation is blocked until the node is not on-line. The code of the two superprocesses is in Figure 2.

```
ConnectionSP =
   while true do
      in("connect", !c)@self.
      add(c)
   enddo
```

```
DisconnectionSP =
   while true do
      in("disconnect", !c)@self.
      rm(c).
      out(ack)@self
   enddo
```

Fig. 2. The code of the superprocesses running at the engineer's node.

Let us now consider process *EngineerProc* that model the engineer's activity. Initially, process *EngineerProc* calls for a connection to cluster $virtualc$ to take place. Eventually, the node goes on-line and *EngineerProc* can download the wanted information from the server and send the mobile process $Search$ for execution at S. Then, *EngineerProc* calls for a disconnection from $virtualc$ to take place and, when the node has gone off-line, a stand alone computation is performed. Later, the process requires a reconnection to the community. When the engineer's node goes back into the cluster, process $Search$ can return the results of its search to the engineer by performing traditional **out** operations at the engineer's node. Such operations have been suspended during disconnection. Before requiring disconnection for working in isolation, *EngineerProc* establishes a communication with one of the other nodes (s_1) by means of operation $\mathbf{out}(t_1)@s_1$ and waits for a tuple matching t_2 at `self`. Finally, after reconnection, *EngineerProc* uploads his work by means of $\mathbf{out}(t_f)$. The schema for process *EngineerProc* is in Figure 3.

```
EngineerProc =
    out("connect", virtualc)@self.
    in(t)@S. eval(Search)@S.
    out("disconnect", virtualc)@self. in(ack)@self
    ... Stand alone computation ...
    out("connect", virtualc)@self.
    out(t_1)@s_1. in(t_2)@self.
    out("disconnect", virtualc)@self. in(ack)@self
    ... Stand alone computation ...
    out("connect", virtualc)@self.
    out(t_f)@S.
    out("disconnect", virtualc)@self. in(ack)@self
```

Fig. 3. The schema for process *EngineerProc*.

5 Translation in standard Klaim and implementation

We conclude the paper by first pointing out a sketch of a possible translation of the extended language into standard Klaim, and then illustrating how the language extensions will be accommodated in the existing Klaim implementation.

In the translation, every cluster c in the net becomes a node with site s_c; moreover, if a node s belongs to c then the tuple (s) is inserted in the tuple space at s_c and the tuple $(cluster, s_c)$ is inserted in the tuple space at s. As regards the operations acting over clusters, **newc** is simply translated into **newloc**. Operation $\mathbf{add}(c)$ performed at s is translated into a sequence of operations that first add the tuple (s) to the tuple space at s_c, then add the tuple $(cluster, s_c)$ in the tuple space at s. More precisely, $\mathbf{add}(c)$ is rendered as

out(`self`)@s_c.**out**(*cluster*, s_c)@`self`. Similarly, operation **rm**(c) is translated into the sequence **in**(`self`)@s_c.**in**(*cluster*, s_c)@`self`.

Standard Klaim communication and migration operations with the new semantics illustrated in Section 3 are translated into sequences of operations that first test for the presence of a cluster that permits the connection and then perform the Klaim operation. Finding a cluster that permits the connection means finding in the local tuple space a tuple of the form (*cluster*, s) such that the tuple space at s contains the address of the node where the Klaim operation should act. While conceptually simple, in practice this search requires a more sophisticated management of tuples of the form (*cluster*, s): a counter should also be used for being able to exhaustively examine all the clusters (and, then, their tuple spaces) to which a node belongs. Timeouts are also used to avoid blocking a process while it searches the tuple space of a cluster node.

It is possible to show that the original semantics can simulate all the computations of the extended semantics. Of course, the converse is not true because the translation does not preserve atomicity (e.g. deadlocks are not preserved too).

As regards the implementation, clusters can be smoothly accommodated in the existing Klaim implementation[1], without making use of the translation sketched above. Indeed, a Klaim net is implemented through a server where nodes must register by using their site. The server allows nodes to communicate both directly or indirectly (i.e. messages pass through the server) and it can be considered as a cluster. Hence, extending the implementation consists in having more than one server in the same net and in allowing a node to register in more than one server. Currently, the extended implementation is in progress.

6 Related Work and Conclusions

The paradigm that is closer to ours is the Ambient calculus [4]. However, the aim of clusters and their features are quite different from ambients' ones. Clusters represent communication capabilities, rather than real physical environments (indeed clusters are more similar to channels). Hence, differently from ambients, clusters can overlap so that, e.g., shared nodes can act as gateways between different clusters. Moreover, clusters cannot move and cannot be nested (in the sense that a cluster cannot occur inside another one), while

[1] The implementation of Klaim consists of Klava, a Java package that provides the run-time system for Klaim operations, and X-Klaim [2], a programming language that extends Klaim with a high level syntax for processes: it provides variable declarations, Klaim operations, assignments, conditionals, sequential and iterative process composition. A compiler is also provided, which translates X-Klaim programs into Java programs that use Klava. X-Klaim syntax and software can be found on-line, at the Klaim site (`http://music.dsi.unifi.it`).

these are the main features of ambients. Similar considerations also hold for the Seal calculus [12].

Another related paradigm is Lime [11]. In Lime, each mobile agent has its own tuple space and all the agents running over an host share the transient tuple space formed by the union of the tuple spaces of each agent. Lime hosts can resemble our clusters, however, in Klaim, clusters can overlap and nodes' tuple spaces are always distinct, thus can be explicitly addressed.

Clusters have turned to be a powerful abstraction for modelling node connectivity, a key aspect of dynamically evolving networks. Clusters can be used to model many real situations in a natural way and to easily express a number of basic properties of node connectivity, such as, e.g., locality and distribution of nodes, efficiency, and fault tolerance. We plan to integrate our current framework with routing information as in [3], in order to be able to hierarchically structure clusters and to have a fine grain control over the routing process of messages. We also plan to extend our framework in order to be able to bound quality of service parameters to clusters, such as, e.g., bandwidth and latency, and to quantify the performance that the connection architecture can guarantee. Finally, we also want to extend the Klaim access control type system [7,6] in order to be able to restrict operations over clusters.

References

[1] Arnold, K., J. Gosling and D. Holmes, "The Java Programming Language," Addison-Wesley, 2000, 3rd edition.

[2] Bettini, L., R. De Nicola, G. Ferrari and R. Pugliese, *Interactive Mobile Agents in* X-Klaim, in: P. Ciancarini and R. Tolksdorf, editors, *Proc. of the 7th Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)* (1998), pp. 110–115.

[3] Bettini, L., M. Loreti and R. Pugliese, *Structured Nets in* Klaim, in: *Proc. of ACM SAC 2000, Special Track on Coordination Models, Languages and Applications* (2000).

[4] Cardelli, L. and A. Gordon, *Mobile ambients*, in: *Foundations of Software Science and Computation Structures (FoSSaCS'98)*, number 1378 in LNCS (1998), pp. 140–155.

[5] De Nicola, R., G. Ferrari and R. Pugliese, Klaim*: a Kernel Language for Agents Interaction and Mobility*, IEEE Transactions on Software Engineering **24** (1998), pp. 315–330.

[6] De Nicola, R., G. Ferrari and R. Pugliese, *Types as Specifications of Access Policies*, in: J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, number 1603 in LNCS (1999), pp. 117–146.

[7] De Nicola, R., G. Ferrari, R. Pugliese and B. Venneri, *Types for Access Control*, Theoretical Computer Science **240** (2000), pp. 215–254.

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

[9] Milner, R., “Communication and Concurrency,” Prentice Hall, 1989.

[10] Milner, R., *The polyadic π-calculus: a tutorial*, Technical Report ECS-LFCS-91-180, Dep. of Comp. Sci., Edinburgh Univ. (1991).

[11] Picco, G., A. Murphy and G.-C. Roman, Lime*: Linda Meets Mobility*, in: D. Garlan, editor, *Proc. of the 21st Int. Conference on Software Engineering (ICSE'99)* (1999), pp. 368–377.

[12] Vitek, J. and G. Castagna, *Seal: A Framework for Secure Mobile Computations*, in: *Internet Programming Languages*, number 1686 in LNCS, Springer, 1999 .

[13] White, J. E., *Telescript Technology: The Foundation for the Electronic Marketplace*, White paper, General Magic, Inc., Mountain View, CA (1994).