Calculi for Network Aware Programming (Extended Abstract)

GianLuigi Ferrari[1] Rosario Pugliese[2] Emilio Tuosto[1]

[1]Dipartimento di Informatica, Università di Pisa
e-mail: {giangi,etuosto}@di.unipi.it

[2]Dipartimento di Sistemi e Informatica, Università di Firenze
e-mail: pugliese@dsi.unifi.it

1 Overview

Highly distributed networks have now become a common infrastructure for many applications which use network facilities to access remote resources and services. *Network awareness*, namely the ability of dealing with *dynamic* changes of network environment, has emerged as a key design principle for wide-area distributed applications. Network-aware computing has prompted the study of the foundations of programming languages with advanced features including mechanisms for agent mobility and for coordinating and monitoring the use of resources. A main challenge is to exploit foundational calculi to identify which programming abstractions are most suitable for network-aware programming. The goal of this paper is to classify and evaluate a number of foundational calculi for network-aware computing (Dπ [13], `Djoin` [8, 9], KLAIM [6], `Ambient` [2, 3, 1]). The benefits and drawbacks of each calculus and its appropriateness to express metaphors for network-aware computing are evaluated along two different guidelines: the programming abstractions the calculus suggests and the underlying programming model. This evaluation will help in understanding the potentials and the advantages of using foundational calculi in the design of new programming languages for network-aware computing.

2 Distributed π-calculus

The Distributed π-calculus [13] (Dπ for short) extends the π-calculus [15] with explicit locations, located channels and with primitives for process mobility. Locations reflect the idea of having administrative domains and located channels can be regarded as channels under the control of certain authorities. The *syntax* of the calculus is illustrated in Table 1. To improve readability, we use a, b, ... as channel names, and h, ℓ, ... as location names; we use e, f, ... when the distinction does not play any role. Intuitively, a *system* consists of a set of agents running independently in parallel. A system is an *allocated thread* $\ell[p]$, where the scope of some channel and location names can be restricted. *Threads* are essentially polyadic π-calculus processes that can additionally create new locations or names ($(\nu\, e)p$) and migrate to other locations ($\ell :: p$). In Dπ tuples of allocated channels and location names can be transmitted over channels. To give a flavour of the Dπ programming model we show the code of a mobile counter, *Cnt*, that initially runs at location h. The counter is initialized to value n and uses the global names *inc*, *dec*, *val* and *jmp* to interact with the environment:

$$Cnt \triangleq h[(\nu a)(\nu\, i)(\overline{a}\langle h\rangle.\overline{i}\langle n\rangle \mid !a(\ell).i(x).\ell :: (\nu\, c)(\overline{c}\langle x\rangle \mid C))]$$

$$\begin{array}{rl} C \triangleq & !inc(z[y]).c(x).\overline{c}\langle x+1\rangle.z :: \overline{y}\langle\rangle \\ & |!dec(z[y]).c(x).(\overline{c}\langle x-1\rangle \mid z :: \overline{y}\langle\rangle) \\ & |!val(z[y]).c(x).(\overline{c}\langle x\rangle \mid z :: \overline{y}\langle x\rangle) \\ & \mid jmp(\ell').c(x).h :: \overline{a}\langle\ell'\rangle.\overline{i}\langle x\rangle \end{array}$$

The counter can be regarded as an *abstract object*: a process willing to interact with *Cnt* can only use the interface, i.e. the global channel names *inc*, *dec*, *val* and *jmp*, and has no control over the private channels a and i. Clients can require standard services (i.e. *inc*, *dec* and *val*) by transmitting an allocated channel ($z[y]$); *Cnt* will return the result of the service at the channel y located at z. Clients can also ask the counter to move to a different location by sending the destination location at channel *jmp*.

A possible client for the counter is the process

$$m[h :: \overline{val}\langle m[v]\rangle \mid Q]$$

The client requires the service by sending a process asking for the service to the counter location. The counter will send the result by spawning the mobile process $m :: \overline{v}\langle n'\rangle$ which moves itself to the client location to deliver the value n' of

the counter. It is important to notice that in Dπ to access a remote resource (e.g. the counter) one has to know where it is located.

3 Distributed `join-calculus`

The `join-calculus` [8] is an "extended subset" of π-calculus calculus which combines the three operators for input, restriction and replication into a single operator, called *definition*, that has the additional capability of describing atomic *joint* reception of values from different communication channels. The Distributed `join-calculus` (`Djoin` for short) [9] adds abstractions to express process distribution and process mobility. The *syntax* of the calculus is given in Table 2. Basic values v can be both channel names x and location names a, location names can be exchanged in communications. We will use ψ and φ to denote finite strings of location names, i.e. elements of $\mathcal{L}^*$. A net is a multiset of located solutions. A solution $\mathcal{D} \vdash_\varphi \mathcal{P}$ consists of a location label φ, of a multiset of running processes $\mathcal{P}$ and of a multiset of active rules $\mathcal{D}$, which define the possible reductions of processes. A process may send an asynchronous message on a name, define new names and reaction rules, fork in parallel components and move its execution to another location using **go**. A definition D is composed of some reaction rules, $J \triangleright P$, and location constructors separated by the $\wedge$ operator. The definition $J \triangleright P$ triggers the execution of the guarded process P when a join-pattern J is recognized. As an example of `Djoin` programming we provide the implementation of a *counter* process. The code of a process with a local definition of *cnt* is:

$$
\begin{array}{l}
\textbf{def}\ cnt\langle x, k\rangle \triangleright \\
\quad \textbf{def}\ count\langle n\rangle \mid inc\langle k\rangle \triangleright count\langle n+1\rangle \mid k\langle\rangle \\
\quad \wedge\ \ count\langle n\rangle \mid dec\langle k\rangle \triangleright count\langle n-1\rangle \mid k\langle\rangle \\
\quad \wedge\ \ count\langle n\rangle \mid val\langle k\rangle \triangleright count\langle n\rangle \mid k\langle n\rangle \\
\quad \textbf{in}\ count\langle x\rangle \mid k\langle inc, dec, val\rangle \\
\textbf{in}\ \texttt{Client}
\end{array}
$$

A client process gains the ability of accessing the counter by passing it an initial value and a continuation channel (e.g. the client may take the form $\textbf{def} \ldots \textbf{in}\ cnt\langle 5, k\rangle$). The counter may be transformed into a mobile counter by allocating channels inc, dec and val at a new location a. The counter, i.e. channels inc, dec and val, will move as long as location a moves. Location a is a sublocation of the current location of *mob_cnt*, but will become a sublocation of b after moving with **go** $\langle \mathsf{b}\rangle$, before executing $count\langle x\rangle \mid k\langle inc, dec, val\rangle$. The code of a process with a local definition of *mob_cnt* is:

$$
\begin{array}{l}
\textbf{def}\ mob_cnt\langle \mathsf{b}, x, k\rangle \triangleright \\
\quad \textbf{def}\ \mathsf{a}[count\langle n\rangle \mid inc\langle k\rangle \triangleright count\langle n+1\rangle \mid k\langle\rangle \\
\quad \wedge\ \ count\langle n\rangle \mid dec\langle k\rangle \triangleright count\langle n-1\rangle \mid k\langle\rangle \\
\quad \wedge\ \ count\langle n\rangle \mid val\langle k\rangle \triangleright count\langle n\rangle \mid k\langle n\rangle \\
\quad : \textbf{go}\ \langle \mathsf{b}\rangle; count\langle x\rangle \mid k\langle inc, dec, val\rangle] \\
\quad \textbf{in}\ \mathtt{0} \\
\textbf{in}\ \texttt{Client}
\end{array}
$$

4 KLAIM

KLAIM [6] is an asynchronous higher-order calculus which extends the Linda [4, 12] paradigm to distributed and mobile processes. The *syntax* of the calculus is reported in Table 3. We assume the existence of a set of *sites* S and of a set of *site variables*, that includes the distinguished variable `self`. The `self` variable is used by processes to refer to their current execution site. KLAIM processes may perform three different kinds of actions: accessing tuple spaces (i.e. multisets of tuples), spawning processes and creating new nodes in a net. The (non–blocking) operation $\mathbf{out}(t)@u$ adds the tuple resulting from the evaluation of t to the tuple space (TS, for short) located at u. Two (possibly blocking) operations, $\mathbf{in}(t)@u$ and $\mathbf{read}(t)@u$, access tuples in the TS located at u. The operation $\mathbf{in}(t)@u$ evaluates t and looks for a matching tuple t' in the TS at u; if such a t' exists, it is removed from the TS. The corresponding values of t' are then assigned to the variables in the formal fields of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available. The operation $\mathbf{read}(t)@u$ differs from $\mathbf{in}(t)@u$ only in that the matching tuple t' is not removed from the TS at u. New threads of executions are dynamically activated through the operation $\mathbf{eval}(P)@u$ that spawns a process (whose code is given by P) at the node named u. New nodes in a net can be created through the operation $\mathbf{newloc}(\widetilde{u})$ and then accessed via the site variables $\widetilde{u}$. This operation is not indexed with a site identifier because it is always executed at the current execution node.

Nets are collections of nodes. A *node* is a term of the form $s ::_\rho P$, where the site s is the node address, P gives the processes running at s, and ρ is the *allocation environment*, namely a function mapping site variables to sites. ρ links the locality variables which occur free in P to certain sites. The idea is that allocation environments act as proxy mechanisms of the nodes in the net. Processes have not direct access to nodes and can get knowledge of a site either through their (local) allocation environment or through communication with other processes (which, again, exploits other allocation environments).

Systems	P	::=	$\mathtt{0}$	*Null system*
		|	$P \mid Q$	*Composition*
		|	$(\nu\, e)P$	*Restriction*
		|	$\ell[p]$	*Allocated thread*
Threads	p	::=	$\mathtt{0}$	*Null process*
		|	$p \mid q$	*Composition*
		|	$(\nu\, e)p$	*Restriction*
		|	$u :: p$	*Migration*
		|	$\overline{u}\langle U\rangle.p$	*Output*
		|	$u(X).p$	*Input*
		|	$!p$	*Replication*
		|	**if** $u = v$ **then** p **else** q	*Conditional*

Table 1. $\mathrm{D}\pi$ **syntax**

Nets	S, T	::=	$\mathcal{D} \vdash_{\varphi} \mathcal{P}$	*Located solution*
		|	$S \parallel T$	*Distributed CHAM*
Processes	P, Q	::=	$x\langle\widetilde{v}\rangle$	*Asynchronous message*
		|	**def** D **in** P	*Local definition*
		|	$P \mid Q$	*Parallel composition*
		|	**go** $\langle \mathsf{a}\rangle; P$	*Migration request*
		|	0	*Null process*
Definition	D, E	::=	$J \triangleright P$	*Elementary clause*
		|	$\mathsf{a}[D : P]$	*Location constructor*
		|	$D \wedge E$	*Simultaneous definition*
Join patterns	J, J'	::=	$x\langle\tilde{v}\rangle$	*Asynchronous reception*
		|	$J \mid J'$	*Joining messages*

Table 2. `Djoin` **syntax**

Nets	N	::=	$s ::_{\rho} P$	*Single node*
		|	$N_1 \parallel N_2$	*Net composition*
Processes	P	::=	$\mathtt{0}$	*Null process*
		|	$a.P$	*Action prefixing*
		|	$P_1 \mid P_2$	*Process composition*
		|	X	*Process variable*
		|	$A\langle\widetilde{V}\rangle$	*Process invocation*
Actions	a	::=	$\mathbf{out}(t)@u$	*Output*
		|	$\mathbf{in}(t)@u$	*Input*
		|	$\mathbf{read}(t)@u$	*Read*
		|	$\mathbf{eval}(P)@u$	*Process creation*
		|	$\mathbf{newloc}(\widetilde{u})$	*Node creation*

Table 3. KLAIM **Syntax**

We now present the KLAIM implementation of a mobile process which is used to collect the mailboxes of a user over distinct nodes (e.g. accounts). The code of the mobile process is:

$$Fwd(id, u_h) \stackrel{\text{def}}{=} \mathbf{in}(''\text{mbox}'', id, !x)@\texttt{self}.\ \mathbf{out}(''\text{mbox-at}'', id, \texttt{self}, x)@u_h.\texttt{0} \mid \mathbf{read}(''\text{remote-mbox}'', id, !u)@u_h.\ \mathbf{eval}(Fwd(id, u_h))@u.\texttt{0}$$

Process Fwd takes as parameters the identificator id and the home address u_h of the user. When executed, it withdraws the id's mailbox at the current execution site and sends it to address u_h; concurrently, it looks for the next site u to visit and, then, spawns a copy of itself at u.

5 Ambient Calculus

The Ambient calculus [2] relies on the notion of *ambient* that can be thought of as a bounded environment where processes cooperate. An ambient has a name, a collection of local agents and a collection of subambients. Ambients can be moved as a whole under the control of agents; these are confined to ambients. The *syntax* of the calculus is reported in Table 4. Ambient processes use *capabilities* for controlling interaction. Indeed, by using capabilities, an ambient can allow other ambients to perform certain operations over it without having to reveal its actual name (which would give a lot of control over it). A name n is a capability to enter, exit or create a new copy of an ambient named n. Capability *in* M serves for entering into ambient M, *out* M for exiting out of M and *open* M for opening up M. The possession of one or all of these capabilities is insufficient to reconstruct the original ambient name from which they were extracted. Multiple capabilities can be combined into paths. Process $n[P]$ is an ambient with name n and process P running inside. Nothing prevents the existence of two or more ambients with the same name. Process $M.P$ executes the action corresponding to capability M and then behaves like P. Communication is asynchronous and anonymous (no process or communication channel is explicitly referred), and takes place locally within a single ambient. The objects that can be communicated are ambient names and capabilities, that may be thought of as rights to commit some operations on ambient names.

As an example of Ambient programming, consider the case of a process which wants to enter a "secure" ambient, i.e. an ambient whose name w is restricted. The following program (borrowed from the firewall example of [2]) describes the protocol, based on passwords k and k', that allows Q to enter w. The third name k'' is necessary to confine Q thus preventing it to interfere with the protocol.

$$(\nu w)(w[k[\textit{out } w.\textit{in } k'.\textit{in } w] \mid \textit{open } k'.\textit{open } k''.P]) \mid k'[\textit{open } k.k''[Q]]$$

6 Evaluation

The basic abstractions of Dπ and Djoin are locations and trees of locations, respectively. Locations are first class entities and can be viewed as the addresses of interpreters running on hosts. Despite of the similarities, there are some differences on the way locations are exploited. In fact, Dπ localities are mainly adopted to program migration and to define allocated channels, i.e. as a way to model the distributed object invocation mechanisms. Furthermore, a process which needs a non-local name (e.g. a remote resource), say a, has to know the location where a lays and has to migrate to the location of a in order to exchange values over a. In Djoin, instead, there is a more structured location concept: a location is a tree composed by the root location and its sub-locations. When a process defined on a moves itself to another site, the whole tree rooted at a moves with the process. A second difference is that Djoin locations are not necessary in determining the allocation site of remote channels. It is the communication infrastructure (uniqueness of join receptor) that worries about determining the service channel site.

The synchronization mechanism of Dπ is local synchronous communication and extends that of the π-calculus with the possibility of exchanging allocated channels. The synchronization mechanism of Djoin is more refined: it is based on join patterns and allows two processes to synchronize on a set of channels (those channels defined in the pattern). The uniqueness of join receptor and the join pattern construct confer to the Djoin programming model a "functional" feature. An experimental implementation of Djoin called JoCaml, is available at http://pauillac.inria.fr/jocaml/.

KLAIM basic abstractions are sites, located tuple spaces and the net coordination mechanism. Sites are first class entities and, basically, corresponds to KLAIM interpreters. Located tuple spaces provide both a form of synchronization mechanism and a tool for interprocess communication. Processes, which are the computational entities, can interact by asynchronous distributed object method invocations. The Linda-like communication paradigm is anonymous and associative and seems to be more appropriate than the one based on *naming*, (e.g., Dπ and Djoin), to coordinate (possibly) heterogeneous mobile agents applications for open systems [5]. Notice that this communication mechanism has been adopted by the Java API [16]

Processes	P, Q	::=	$(\nu\, n)P$	*Restriction*
		|	$\mathtt{0}$	*Inactivity*
		|	$P \mid Q$	*Composition*
		|	$!P$	*Replication*
		|	$M[P]$	*Ambient*
		|	$M.P$	*Capability action*
		|	$(x).P$	*Input action*
		|	$\langle M \rangle$	*Output action*
Capabilities	M	::=	x	*Variable*
		|	n	*Name*
		|	*in* M	*Can enter* M
		|	*out* M	*Can exit* M
		|	*open* M	*Can open* M
		|	ε	*Empty path*
		|	$M.M$	*Path*

Table 4. `Ambient` **syntax**

The net coordination mechanism heavily relies on the notion of allocation environments that give local meaning to network references. As it happens for Dπ and `Djoin`, whenever a process knows a site s it may access to the services granted at s. A KLAIM process does not need to migrate on a TS site to access the TS contents, i.e. communications may be remote. KLAIM provides a platform to program network services where the set of available services (coded as tuples in the tuple spaces) at any given moment may dynamically change. This mechanism is very close to the one provided by some commercial platforms, such as, e.g., Jini [7]. An experimental implementation of KLAIM, called X-KLAIM, can be downloaded at `http://rap.dsi.unifi.it/klaim.html`.

The key notion of the `Ambient` calculus is that of *ambient*. An ambient can be thought of as a bounded environment where processes cooperate. An ambient has a name (and the only way of using them is through explicit naming), a collection of local agents and a collection of subambients. Ambients can be moved as a whole under the control of agents; these are confined to ambients[1]. The major novelty of the ambient programming model is given by the movement of self contained nested ambients (that include data and computation) as opposed to the previous paradigms that move single data or processes.

Ambient operations (capabilities) provide a full fledged coordination language to move, compose and rearrange ambient structure. Mobile computational ambients can be seen as *wrappers* of software libraries as well as administration domains. Indeed, the interpretation of ambients as wrappers has been exploited in [10] to map the `Ambient` in `JoCaml`. There, an ambient is modeled as a black-box with an interface towards other ambients in the environment.

We can think of ambient names as being an abstraction of localities. Differently from the calculi of previous sections, in `Ambient` the knowledge of the name of a location is not enough to access its service: it is necessary to know the route to the location. Interaction is local, anonymous and asynchronous. An experimental implementation of the ambient calculus in the `JoCaml` language is described in [10]. The implementation consists of a translation of ambients in `JoCaml`.

Table 5 summarizes our evaluation (we refer to the full paper for more details).

[1] A KLAIM node is much like an ambient but without subambients.

References

[1] L. Cardelli, G. Ghelli, A. Gordon. Mobility Types for Mobile Ambients. In *Proc. of ICALP'99* (J. Wiedermann, P. van Emde Boas, M. Nielsen, Eds.) *LNCS* 1644, pp.10-24, Springer, 1999.

[2] L. Cardelli, A. Gordon. Mobile Ambients. In *Proc. of FoSSaCS'98* (M. Nivat, Ed.), *LNCS* 1378, pp.140-155, Springer, 1998.

[3] L. Cardelli, A. Gordon. Types for Mobile Ambients. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pp.79-92, ACM Press, 1999.

[4] N. Carriero, D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444-458, ACM Press, 1989.

[5] G. Cabri, L. Leonardi, F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In *Proc.*

	Dπ	Djoin	KLAIM	Ambient
Communication	local channel-based explicit naming synchronous	local channel-based explicit naming asynchronous	local/remote tuple-based anonymous/associative asynchronous	local message-based anonymous asynchronous
Mobility	process/agent	process/code	code/process/agent	agent
First-class objects	channels/locations	channels/locations	processes/sites	capabilities and ambient names
Experimental Implementation		JoCaml Objective-Caml	X-KLAIM KLAVA	JoCaml Ambit
Administrative Domains	flat model locations	hierarchical model locations tree	flat model nodes and allocation environments	hierarchical model ambients
Coordination Mechanism	located threads and located channels	local processes and local definitions	allocation environments and nodes	ambients

Table 5. Calculi for Network-Aware Programming: An Assessment

of the 2nd Int. Workshop on Mobile Agents (K. Rothermel, F. Hohl, Eds.), *LNCS* 1477, pp.237-248, Springer, 1998.

[6] R. De Nicola, G. L. Ferrari, R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility, *IEEE Transactions on Software Engineering*, 24(5):315-330, IEEE Computer Society, 1998.

[7] K. Edmonds. *Core Jini*. Prentice Hall, 1999.

[8] C. Fournet, G. Gonthier. The reflexive CHAM and the join calculus. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pp.372-385, ACM Press, 1996.

[9] C. Fournet, G. Gonthier, J.J Lévy, L. Maranget, D. Rémy. A Calculus of Mobile Agents, In *Proc. of CONCUR'96* (U. Montanari, V. Sassone, Eds.), *LNCS* 1119, pp. 406-421, Springer, 1996.

[10] C. Fournet, J.J Lévy, A. Schmitt. A Distributed Implementation of Ambients, Available at `http://join.inria.fr/ambients.html`.

[11] D. Garlan, D. Le Metayer (Eds). *Coordination Languages and Models. LNCS* 1282, Springer, 1997.

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

[13] M. Hennessy, J. Riely. Resource Access Control in Systems of Mobile Agents. In *Proc. Int. Workshop on High–Level Concurrent Languages*, vol.16(3) of *Electronic Notes in Theoretical Computer Science*, Elsevier, 1998.

[14] D. Le Metayer, Ed. *Theoretical Computer Science*, 240(1), special issue on Coordination, Elsevier Science, July 2000, 1998.

[15] R. Milner, J. Parrow, D. Walker. A calculus of mobile processes, (Part I and II). *Information and Computation*100:1-77, Academic Press, 1992.

[16] Sun Microsystems. The JavaSpace Specifications. `http://java.sun.com/products/javaspaces`, 1997.