From Process Algebra to Java Code

Andrew Phillips, Susan Eisenbach and Daniel Lister
Department of Computing, Imperial College
London SW7 2BZ
email: anp, sue, del98@doc.ic.ac.uk

Abstract

The $\delta\pi$-calculus, a calculus based on the π-calculus, is a model for mobile distributed computation. The $\delta\pi$-calculus can be used to specify applications, in order to reason about their security and correctness properties. The $\delta\pi$ primitives have been been implemented as a Java API. The implementation in Java provides a means of bridging the gap between application specification and implementation.

1 Introduction

The Internet has grown substantially in recent years, and an increasing number of applications are now being developed to exploit this distributed infrastructure. Mobility is an important paradigm for such applications, where mobile code is supplied on demand and mobile components move freely within a given network. However, mobile distributed applications are notoriously difficult to develop. Not only do they involve complex parallel interactions between multiple components, but they must also satisfy strict requirements for security, reliability and correctness. One could argue that the development of such applications requires a means of understanding and reasoning about mobile distributed computation in a rigorous manner, through the use of an appropriate *model of computation*.

This paper presents the $\delta\pi$-calculus, as a model for mobile distributed computation. The term *mobility* refers to the movement of *code* and also of *running processes*. The term *distributed* is used in the broad sense, to refer to computation over a wide-area network. The paper shows how $\delta\pi$ can be used to specify simple applications, in order to reason about their security and correctness properties. To make this model available to programmers, it has been implemented as a Java API. The paper discusses both implementation and use.

2 Model of Computation

2.1 Requirements and Approach

The requirements of a good model for mobile distributed computation can be summarized as follows. The model should be *expressive* enough to capture the essential properties of mobile distributed computation, in order to describe a wide range of applications. It should be *rigorous*, to allow reasoning about the properties of these applications and it should be *concise*, remaining as simple as possible and avoiding ambiguity. It should also be at the right level of *abstraction*, bearing a close correspondence with the basic forms of computation it is attempting to describe.

There are a number of essential properties[3, 4] that a model for mobile distributed computation should be able to express. Distributed computation is **concurrent** by definition, since it involves the parallel execution of processes on different machines within a network. A means of expressing the **location** of processes is required, not only in relation to physical machines but also to logical domains. **Communication** needs to be modelled within and amongst these locations, which may be spread out over a wide-area network. **Failure** and **delay** are two inherent properties of distributed systems. These are often indistinguishable, giving rise to the notion of **locality**, which states that global synchronization across a network is impossible. **Mobility** is another important property of distributed computation, including *code mobility*, used in remote procedure calls and Java^{TM} applets, and *process mobility* used in mobile software agents or mobile hardware devices. **Security** mechanism are also required to protect shared *data* and *services,* to safeguard *applications* against malicious attack, and to *regulate movement* to and from locations.

A promising approach to modelling computation is to develop a suitable calculus. Calculi can be thought of as very simple programming languages, which provide a concise description of computation that facilitates rigorous analysis. They have a precise syntax and a computable operational semantics, which are both formally defined, together with a computation state that is implicit in the terms of the calculus. This contrasts with many alternative models of computation, including most automata models, where the state needs to be given explicitly as a separate component, and where states and transitions are often informally described. Calculi have been used successfully for many years to model various forms of computation. An important example is the lambda-calculus, which was developed to model functional computation. More recently, the π-calculus of Milner, Parrow and Walker[10] has been successfully developed as a model for concurrent computation.

2.2 The Asynchronous π-calculus

It has been argued[14] that an *asynchronous* choice-free variant of the π-calculus, first proposed by Honda and Tokoro[7], is a suitable foundation on which to build a model for mobile distributed computation. The asynchronous π-calculus pro-

$$\frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

$$\frac{P \longrightarrow Q}{P|R \longrightarrow Q|R}$$

$$\frac{P \longrightarrow Q}{\nu x\, P \longrightarrow \nu x\, Q}$$

$$x < z > |x(y) = P \longrightarrow P_{\{z/y\}}$$

$$\begin{array}{rcl} P|() & \equiv & P \\ P|Q & \equiv & Q|P \\ P|(Q|R) & \equiv & (P|Q)|R \\ \nu x\,() & \equiv & () \\ \nu x\, \nu y\, P & \equiv & \nu y\, \nu x\, P \\ \nu x\, P & \equiv & \nu y\, P_{\{y/x\}},\ y \notin FV(P) \\ P|(\nu x\, Q) & \equiv & \nu x\, (P|Q),\ x \notin FV(P) \\ x(y) = P & \equiv & x(z) = P_{\{z/y\}},\ z \notin FV(P) \\ !P & \equiv & P|!P \end{array}$$

Figure 1: π-calculus semantics

vides a complete, concise and rigorous description of concurrent computation and offers a good level of abstraction for reasoning about concurrent applications. This has been demonstrated by the PICT programming language [12], in which a wide range of concurrent applications have been developed. In addition, channels in the π-calculus bear a close resemblance to many entities in distributed computing, including IP addresses, communication channels, remote references and cryptographic keys. They can be used to model *data* including passwords, keys and references to documents or other media files, and *services* such as applications, procedure calls, system calls, or access to printers and other hardware resources. Furthermore, asynchronous communication in the π-calculus is close to reliable datagram communication, which lies not far above the Internet Protocol (IP). Finally, a substantial body of research has been conducted on the π-calculus, including type systems, encodings and theories of equivalence. By using the π-calculus as the basis for a model for distributed computation, much of this associated theory can be re-used.

The asynchronous π-calculus is described below in terms of *processes* P, Q where the set of *variables* is ranged over by x, y, z.

$$P, Q ::= () \mid P|Q \mid \nu x\, P \mid x < z > \mid x(y) = P \mid !P$$

The Null process $()$ does not perform any computation. Parallel Composition $P|Q$ executes processes P and Q in parallel. Restriction $\nu x\, P$ declares a new communication channel x, known only to P. Output $x < z >$ sends a value z on channel x. Input $x(y) = P$ receives a value on channel x and assigns it to the variable y in process P, which continues executing. Replication $!P$ behaves like an infinite number of copies of process P. The operational semantics of the π-calculus is summarized in Figure 1, in terms of structural congruence $\equiv$ and reduction $\longrightarrow$, where $P_{\{z/y\}}$ is the result of substituting all free occurrences of y with z in P. The asynchronous π-calculus is an excellent model of concurrent computation that can also express many distributed concepts. It cannot, however, express the notion of location, which is fundamental to distributed computation.

$$a.x < n > \mid a[Q \mid x(m) = P] \longrightarrow a[Q \mid P_{\{n/m\}}]$$
$$x(m) = P \mid a[Q \mid ..x < n >] \longrightarrow P_{\{n/m\}} \mid a[Q]$$
$$b[P \mid \uparrow a\, Q] \mid a[R \mid +b\, S]] \longrightarrow a[R|S \mid b[P|Q]]$$
$$a[b[P \mid \downarrow a\, Q] \mid R \mid -b\, S]] \longrightarrow b[P|Q] \mid a[R|S]$$

$$\frac{P \longrightarrow Q}{a[P] \longrightarrow a[Q]}$$

$$\begin{array}{rcl} \nu n\, a[P] & \equiv & a[\nu n\, P],\ a \neq n \\ a[()] & \equiv & () \end{array}$$

Figure 2: $\delta\pi$-calculus semantics

2.3 The $\delta\pi$-calculus

The $\delta\pi$-calculus extends the asynchronous π-calculus with process terms for describing locations or *agents*, which can be thought of as bounded regions of computation. It also introduces processes for expressing communication between agents and the movement of agents relative to each other. The $\delta\pi$-calculus is described below in terms of *processes* P, Q. The set of channel variables is ranged over by x, y, z as in the π-calculus, and the set of agent variables is ranged over by a, b, c. *Generic* variables m, n can be either channels or agents.

$$\begin{array}{rcl} P, Q ::= & () \mid P|Q \mid \nu x\, P \mid x < z > \mid x(y) = P \mid !P \mid \\ & a[P] \mid a.x < n > \mid ..x < n > \mid \uparrow a\, P \mid \downarrow a\, P \mid +a\, P \mid -a\, P \end{array}$$

Agent Definition $a[P]$ defines an agent with *identity* a and *body* P. Agents form a tree structure where each agent has a single parent and zero or more child agents. For example, in the expression $b[Q \mid a[P \mid c[R] \mid d[S]]]$ agent a has body P, parent b and two children c and d. The parent of an agent is also called its *location*. Child Output $a.x < n >$ sends a value n on channel x to child agent a. Parent Output $..x < n >$ sends a value n on channel x to the parent agent. Enter $\uparrow a\, P$ moves the enclosing agent inside a neighbouring agent a and then continues with process P. For security reasons, one agent cannot enter another without permission. Leave $\downarrow a\, P$ moves the enclosing agent out of its parent. For security reasons, an agent cannot leave its parent without permission. In both cases, when an agent moves to and from a location it takes with it all internal computation, including any child agents. Accept $+a\, P$ allows a neighbouring agent with identity a to enter, and then continues with process P. Release $-a\, P$ allows a child agent with identity a to leave, and then continues with process P.

The operational semantics of the $\delta\pi$-calculus is summarized in Figure 2. It is an extension of the reduction rules and structural congruences of the asynchronous π-calculus, defined previously, with the added property that both channel and agent names can be restricted and communicated over channels.

3 An Example

The following example uses the $\delta\pi$-calculus to model a client, which downloads a mobile application from a remote server. The client and server machines are modeled using *client* and *server* agents, respectively, in parallel with the underlying network infrastructure. The example makes use of *tuples*, and also *process variables* (starting with an upper-case character) that represent processes defined in a top-level environment.

```
( server[!request(client, applet) = ( applet[↓server↑client Service] | -applet())]
| client[νapplet( ..route<server, request, (client, applet)> | +applet User)]
| !route(a, x, m) = a.x<m>
)
```

The **underlying network** routes messages by continually listening on the *route* channel for an agent name, a channel and a message. The message is forwarded to the specified agent along the given channel. The **server** continually listens for requests on the *request* channel. A request is a pair consisting of the identity of a client machine and an applet name. For each request received, the server creates a new agent with the name supplied, and this agent then leaves the server and enters the client where it provides some useful service. The **client** creates a secret applet name and sends a request to the server, via the network. In parallel, it waits for an agent bearing this name to arrive and then makes use of the service provided.

Using the formalism of weak bisimulation inherited from the π-calculus, it is possible to define a suitable notion of equivalence $\approx$ in $\delta\pi$ by considering certain reduction steps as *internal*. Assuming that the server and client are unique agents, which can be enforced using an appropriate type system, the correctness of the above example can be proved by the following equivalence, where process variables are used to give a more concise representation:

```
server[Server] | client[ Client] | Router≈
server[Server] | client[applet[Service] | User] | Router
```

The equivalence states that a request from client to server, as outlined in the example, is equivalent to the request being successfully fulfilled. This holds for all *Service* and *User* processes. A loose analogy is to say that a lock and key function correctly if turning the key in the lock is equivalent to opening the door, where the mechanism of the lock itself is considered to be internal.

Security properties can also be verified using the same formalism. The equivalence below states that correctness is preserved in the presence of an attacker. This holds for all possible *Attack* processes.

```
server[Server] | client[ Client] | attacker[Attack] | Router ≈
server[Server] | client[applet[Service] | User] | attacker[Attack] | Router
```

Description	$\delta\pi$	Mapping
Restriction	ν c P	Channel c = restrict(); P;
Output	$x < n >$	output(x,n);
Input	x(n)=P	Message n = input(x); P;
ChildOutput	$a.x < n >$	childOutput(a, x, n);
ParentOutput	$..x < n >$	parentOutput(x, n);
Accept	+a P	accept(a); P;
Release	-a P	release(a); P;
Enter	$\uparrow$ a P	enter(a);
Leave	$\downarrow$ a P	leave();
Replication	!P	while(true) { P }
Parallel	$P\|Q$	AgentProcess p = new MyAgentProcess(); AgentProcess q = new MyAgentProcess(); parallel(p); parallel(q);
Parallel	$P\|a[...]$	AgentProcess p = new MyAgentProcess(); Agent a = new ChildAgent(); parallel(p); parallel(a);

Table 1: Mapping for constructs in AgentProcess

4 Implementation in Java

4.1 The Java API

For the $\delta\pi$ constructs, there are corresponding methods have been implemented. They are defined in table 1. There is no direct mapping for the $\delta\pi$! construct. The same functionality can be achieved by using a while loop in the main method. The complete method definitions are available from [9]. The most important ones are:

AgentProcess accept, childOutput, enter, leave, output, parentOutput, release

RunnableProcess input, output, restrict

StaticAgent parallel

MobileAgentApplication addLocalProcess, addAgent

4.2 Providing Strong Mobility

Many Java based mobile agent systems only support weak mobility (capture of an agent's *code and object state*)[8, 17]. However, after migration, an agent and its processes should resume execution in exactly the same state at the same code position. The system therefore provides support for strong mobility (capture of agent's code, object state and *control state*).

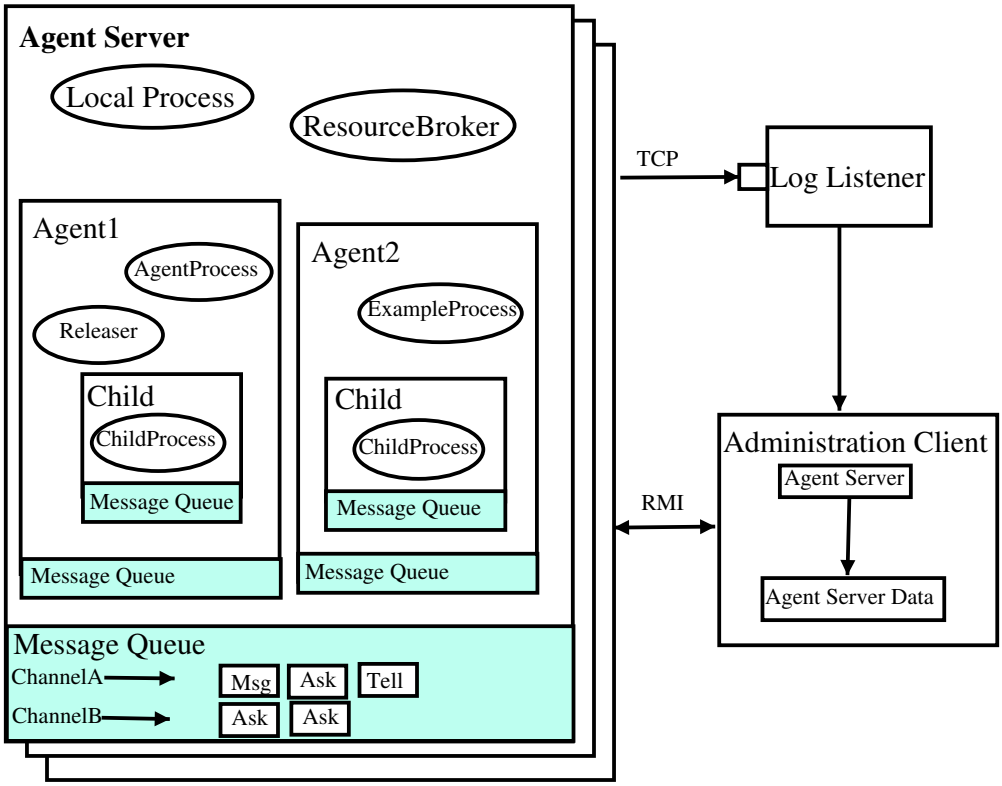

Figure 3: System Architecture

Java does not provide any mechanism for capturing and restoring a thread's state. This could be achieved by modifying the JVM so such functionality is available or by instrumenting code to keep track of the state. Modification of the JVM removes one of the main motivation behind using Java as a basis for a mobile agent system; a widely available virtual machine.

It is possible to capture the state of a running process at the language level, meaning a modified virtual machine is not required. Source is instrumented with code to save runtime information before migration and reestablish it before restart. From the programmers' point of view such systems support transparent migration but the strong mobility is provided on a system with only weak mobility. The code instrumentation can be achieved by means of a preprocessor.

4.3 Architecture

The mobile agent system is formed from multiple agent servers, one for each site, and one or more administration clients (see figure fig:arch:arch. Each agent server has a root location. This contains all the local processes and agents for the site. Each agent has an associated message queue, which stores all the

messages on the appropriate channel for that agent's scope.

An agent server must be run at each site in the distributed system. It is responsible for overseeing the execution of agents and processes at the site and receiving and distributing messages destined for agents at that site. The agent server also deals with requests for information from adminstration clients. It reports the server's current status and provides a tree of the agents and processes executing at the site. The agent server can also provide reports for any of the agents or processes at the site. It streams a copy of the log output for the selected agents or process to a socket on the client machine. Every process, be it local or within the scope of an agent, currently executing at a site is assigned its own thread.

An administration client can be used to monitor multiple agent servers and the agents/processes running on the servers. It can also be used to control the server, for example to stop the server altogether or to allow a given agent to enter the server. The administration client is the entry point for local processes, agents and application into the distributed system. It can be used to dispatch specific local processes or agents to any currently connected agent server so they can start executing. It can also be used to load an agent application and dispatch each part to the agent server at the appropriate site. The GUI of the administration client also provides an interface to the preprocessor so source code can be instrumented before an agent is dispatched to a server.

A Java thread is unable to suspend another thread so agent processes poll to see if a migrate has been requested for the agent. If this is the case the process must save its state and notify the agent that it is at a point where it can be migrated.

The main method of each agent process should be constructed of method calls to pieces of Java computation or method calls for the collaboration and communication constructs of $\delta\pi$. A mark is assigned to each of these method calls to allow for tracking of the execution state.

The check for migration is made after each piece of computation or communication in the main method. Statements are inserted to build a data object which contains the values of all the class and local variables in the main method at that point. The execution mark is also recorded.

After migration, the state is restored by skipping all the previously executed code. Once the next mark (the point after the migration took place) is found, the class and local variables have their values restored from the data object. The process then waits until all other migrated processes have been restored before execution resumes from the correct point.

Before making a method call that will induce a migration, the data object is built and the current execution mark for that process is set. The migration isn't announced to all the other threads until the agent is capable of leaving its parent and the new location is willing to accept it. At that point, all the dependent processes get into a migrateable state and the agent is moved. There is obviously an overhead on application performance due to the extra code that has been inserted. However, this overhead is minimised to a few checks and work is only done when a migration request has been detected.

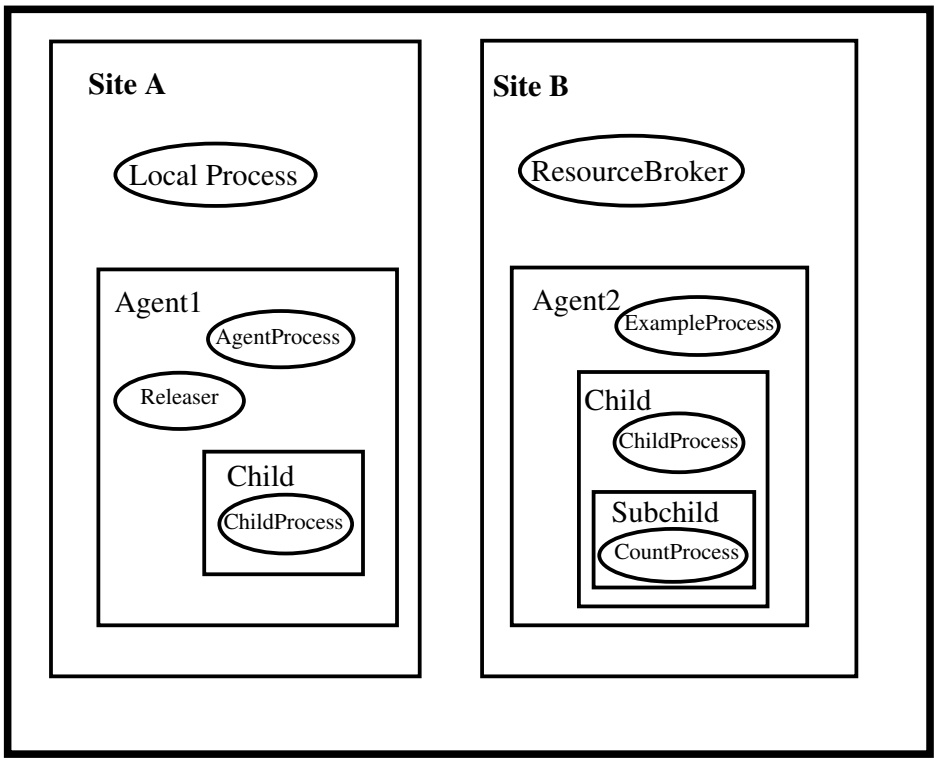

Figure 4: A Mobile Agent Application

Although it would be possible to carry out method splitting to allow for migration at arbitrary points within method calls, this is considered an unnecessary overhead. The number of migration checks that would have to take place would be overwhelming for any real-world application and it would also mean the preprocessing of many more source files (eg libraries). The constraint that an agent can only migrate once all dependent processes have returned to a point in their main method seems the best solution and is in keeping with separating collaboration and communication code from that of computation code. The code instrumentation is achieved by making use of Transmogrify [16] to parse the source and provide an AST which can be manipulated before it is written back out to a file.

4.4 Building a Mobile Agent Application

A programmer can make use of the mobile agent system by just writing local process and agents. The administration client allows for these to be dispatched to any agent server. However, for convenience, an entire mobile agent application can be created. All channels are public, with a public identifier, unless they

are created with the `restrict()` method which will assign a secret identifier.

All mobile agent applications should extend `MobileAgentApplication`. A mobile application is formed by creating instances of local processes and/or agents. Each process or agent must be associated with the site to which it is to be dispatched. The administration client can then be used to load an application and all the local processes and agents will be dispatched to the appropriate agent servers for execution.

All mobile agents should extend the class `MobileAgent`. It is also possible to define a static agent, and in this case it should extend the class `StaticAgent`. Agents are formed from a number of agent processes and may have child agents as well. Agents are formed by calling `parallel(process)` or `parallel(agent)` on created instances of an `AgentProcess` or Agent respectively. All agents are assigned an unique identifier on creation. An agent keeps a reference to its parent, and also the root location in which it is currently executing.

The abstract class `RunnableProcess` implements two interfaces, `Runnable` and `Serializable`. It is the superclass for `LocalProcess` and `AgentProcess`. Because all processes implement `Serializable` they can be transferred across a network in a serialised form. Since the processes implement the `Runnable` interface, upon arrival at a site, the agent server is able to create a new thread for this process and start it executing. Each process must provide a `public void main()` method which defines the work of the process.

All local processes should extend the class `LocalProcess`. Local processes execute within the scope of the root agent at a site. They are unable to migrate but they can communicate along channels with other local processes and top level agents at that site. The main method of a local process can contain anything it wants as long as the process remains serialisable.

All agent processes should extend the class `AgentProcess`. All agent processes must be instrumented by the preprocessor before they can be used in a migrating agent. The main method of an agent process should be constructed of method calls to pieces of Java computation or method calls to the collaboration and communication constructs of $\delta\pi$. Although the programmer isn't constrained to placing the $\delta\pi$ method calls within methods other than main, the separation of collaboration and communication code from that of computation code is desired. The mobile agent may still function correctly if an output was placed within an arbitrary private method. The one constraint that exists now is that any method call, for example `enter(host)`, which can result in a migrate must be located within the `main` method due to the current functionality of the preprocessor.

The applet example has been executed with the *Client* and *Server* processes running on separate remote runtime systems, where routing is performed by the physical network infrastructure. The code can be seen in the Appendix.

5 Conclusion

The $\delta\pi$-calculus appears to be an appropriate model of mobile distributed computation, which can be used to reason about the security and correctness properties of applications. The existence of a Java API based on this model allows the gap between specification and implementation to be bridged with minimal effort.

For the future we have plans to further validate our approach. Ongoing research aims to model more complex and diverse applications. A central server application has also been implemented, where a central server keeps track of the location of a number of registered mobile client agents, allowing them to communicate with each other irrespective of their location. For the $\delta\pi$ models we hope to expand the theory of equivalence. There are also plans to compare the programs written using the $\delta\pi J$ API to programs that accomplish the same tasks written without it.

Related calculi include the Ambient [2], Seal [4], Dpi [13], Nomadic pi Nomadic Pi [18] and Join calculi [5], which express alternative forms of computation to $\delta\pi$.

There are several agent programming systems that support strong mobility. These include Sumatra [1], Ara [11] and Nomads [15] which are implemented by modifying or rewriting the virtual machine and WASP [6] which uses a preprocessor, similar to our approach.

References

[1] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 111–130. Springer-Verlag: Heidelberg, Germany, 1997.

[2] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Proceedings of FoSSaCS '98*, volume 1378 of *LNCS*, pages 140–155. Springer, 1998.

[3] L. Cardelli. Abstractions for mobile computations. *LNCS*, 1603:51–94, 1999.

[4] G. Castagna and J. Vitek. Seal: A framework for secure mobile computations. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, number 1686 in LNCS, pages 47–77. Springer, 1999.

[5] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.

[6] S. Funfrocken. Transparent migration of Java-based moblie agents: Capturing and reestablishing the state of Java programs. In K. Rothermel and F. Hohl, editors, *Mobile Agents: Proceedings of the Second International Workshop*. Springer Verlag, 1998.

[7] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. In M[ario] Tokoro, O[scar] Nierstrasz, and P[eter] Wegner, editors, *Object-Based Concurrent Computing 1991*, volume 612 of *LNCS*, pages 21–51. Springer, 1992.

[8] D.B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[9] Daniel Lister. $\delta\pi J$. http://www.doc.ic.ac.uk/~ del98/agent/index.html, 2002.

[10] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.

[11] Holger Peine and Torsten Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In Radu Popescu-Zeletin and Kurt Rothermel, editors, *First International Workshop on Mobile Agents MA'97*, Berlin, Germany, 1997.

[12] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing. MIT, May 2000.

[13] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Proceedings of POPL '98*. ACM, 1998.

[14] Peter Sewell. Applied π – a brief tutorial. Technical Report 498, Computer Laboratory, University of Cambridge, August 2000.

[15] N. Suri. An overview of the nomads mobile agent system, 2000.

[16] Transmogrify. http://transmogrify.sourceforge.net/, 2002.

[17] Objectspace. http://www.objectspace.com/products/voyager, 1997.

[18] Paweł T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, University of Cambridge, 2000. Also appeared as Technical Report 492, Computer Laboratory, University of Cambridge, June 2000.

A Applet Example

AppletApplication

```
package uk.co.mrlister.applet ; import uk.co.mrlister.agent.* ;
public class AppletAgentApplication extends MobileAgentApplication
{   final static String SERVER_HOST = "server" ;
    final static String CLIENT_HOST = "daniel" ;
    public AppletAgentApplication()
    {   super() ;
        // create instances of the processes and agents
        ServerAgent appletServer = new ServerAgent("AppletServer") ;
        ClientAgent clientA = new ClientAgent
                ("AppletClientA", SERVER_HOST, appletServer.getId()) ;
        // assign the processes and agents to servers
        addAgent(SERVER_HOST, appletServer) ;
        addAgent(CLIENT_HOST, clientA) ;
    }
}
```

ServerAgent

```
package uk.co.mrlister.applet; import java.util.ArrayList ; import
uk.co.mrlister.agent.* ; public class ServerAgent extends
MobileAgent//StaticAgent { public ServerAgent(String name)
    {   super(name) ;
        // add processes
        parallel(new ServerProcess("enterer", this)) ;
    }
    // ------------------------------------------------------------------------
    public class ServerProcess extends AgentProcess
    {   public ServerProcess(String name, Agent agent)
        {  super(name, agent) ; }
        public void main()
        {   Channel request = new Channel("request") ;
            while(true)
            {   Message msg = input(request) ;
                ArrayList msgArgs = (ArrayList) msg.getContent() ;
                String clientSite = (String) msgArgs.get(0) ;
                Long   clientId   = (Long)   msgArgs.get(1) ;
                Agent applet = new AppletAgent("applet", clientSite, clientId) ;
                getAgent().parallel(applet) ;
                getAgent().parallel(
                    new ReleaserProcess("releaser", getAgent(), applet.getId())) ;
            }
        }
    }

    public class ReleaserProcess extends AgentProcess
    {   Long releaseId ;
```

```
        public ReleaserProcess(String name, Agent agent, Long releaseId)
        {   super(name, agent) ;
            this.releaseId = releaseId ;
        }
        public void main()
        {   release(releaseId) ; }
    }
}
```

ClientAgent

```
package uk.co.mrlister.applet; import java.util.ArrayList ; import
uk.co.mrlister.agent.* ; public class ClientAgent extends
MobileAgent //StaticAgent {
    public ClientAgent(String name, String serverHost, Long serverId)
    {   super(name) ;
        // add processes
        parallel(new ClientProcess("requestApplet", this, serverHost, serverId)) ;
    }
    // ---------------------------------------------------------------------
    public class ClientProcess extends AgentProcess
    {   String serverHost ;
        Long   serverId ;
        public ClientProcess(String name, Agent agent, String serverHost, Long serverId)
        {   super(name, agent) ;
            this.serverHost = serverHost ;
            this.serverId   = serverId ;
        }

        public void main()
        {   ArrayList msgArgs = new ArrayList() ;
            msgArgs.add(getRootLocationName()) ;
            msgArgs.add(getId()) ;
            Message msg = new Message(msgArgs) ;
            output(serverHost, serverId, "request", msg) ;
        }
    }
}
```