Verifying Overlay Networks for Relocatable Computations

(or: Nomadic Pict, relocated)

Peter Sewell
University of Cambridge
http://www.cl.cam.ac.uk/users/pes20

Paweł T. Wojciechowski
Poznań University of Technology
http://www.cs.put.poznan.pl/pawelw/

Abstract

In the late 1990s we developed a calculus, Nomadic Pict, in which to express and verify overlay networks, for reliable communication between relocatable computations. Then, efficient system support for relocation was rare, and the calculus was reified in a prototype high-level programming language. Now, relocatable computation is a pervasive reality, though at the level of virtual machines rather than high-level languages. One can ask whether the semantic theory and algorithms developed for Nomadic Pict can be applied (or adapted) to infrastructure for communication between these virtual machines.

1. Introduction

In the 1990s, with the promise of commodity computation dimly visible on the horizon, there was considerable interest in *mobile computation*: systems in which running computations could be moved from one physical machine to another. Many motivating scenarios were put forward, including mangagement of the physical machines (e.g. moving a server away from a machine which needed rebooting), management of network resource (e.g. moving computations 'close' to their communication partners), and managing intermittently connected devices (e.g. moving computations to and from PDAs or laptops). Much of this work was in terms of high-level programming language support for moving computations, and some drew together research on functional programming languages such as Standard ML (Milner et al. 1997), and process calculi such as the π-calculus (Milner et al. 1992). Several languages, including Obliq (Cardelli 1995), Facile (Thomsen et al. 1996), and the Distributed Join Calculus (Fournet et al. 1996), supported not just mobility but also *location-independent* communication between these mobile computations, with distributed infrastructure in the language implementation, which today one might term an overlay network, to reliably deliver messages irrespective of any relocations.

It was clear then that the design of these overlay networks was a challenging problem:

- The distributed algorithms involved are delicate and error-prone, highly concurrent, and with potential races between message delivery and relocation of computations; they are hard to reason about informally.
- The languages cited above have particular algorithms hard-coded into their implementations, but in general the choice of an infrastructure algorithm must be somewhat application-specific: any given overlay algorithm will only have satisfactory performance for some range of migration and communication behaviour; it should be matched to the expected properties (and robustness and security demands) of applications, and of the underlying network.

To address this we developed a small calculus, Nomadic Pict, to permit such algorithms to be described concisely and with mathematical precision (Sewell et al. 1998, 1999; Wojciechowski and Sewell 1999, 2000; Wojciechowski 2001, 2006). The basic primitives included fine-grained concurrency and asynchronous message passing, taken from the π-calculus, together with constructs to create a new named computation (potentially multithreaded), to relocate such a computation from one machine to another, and to send asynchronous messages between these computations. All these are simple to realize, with at most one inter-machine communication required for each transition of the calculus. An overlay network for reliable location-independent communication could then be expressed as a translation of an extended calculus, with that added, into the basic calculus. We implemented the eponymous programming language based on the Nomadic Pict calculi, and experimented with a variety of overlay networks, variously centralised or P2P, with more or less caching, replication, and so on. Our experience was that the level of abstraction of these calculi was a good fit, making it relatively easy to design and understand the overlay network algorithms. Moreover, together with Unyapoth, we developed semantic techniques to support *verification* of the correctness of these algorithms (Unyapoth and Sewell 2001). The key issue here was observational congruence reasoning in the presence of assumptions under which particular computations could be guaranteed (temporarily) *not* to relocate, thus controlling the races between message delivery and relocation.

Ten years later, in 2008, relocatable computation is finally becoming a commonplace reality. This is happening not at the programming-language level we envisioned before, but via checkpointing and movement of virtual machine images, which provides a pervasive (and narrow) API at which to cut the software stack. However, when it comes to looking at communication between virtual machines, this may not be a significant difference. In this position paper we therefore ask whether the Nomadic Pict abstractions could be directly applied, or be adapted, to solve problems in this new setting.

2. The Nomadic π Calculi, Relocated

In this section we recall the Nomadic π calculi, shifting terminology to match the hypothesised virtual machine application.

The main entities are *sites* s and *virtual machines* a. Sites represent physical machines; each site has a unique name. Virtual machines are units of running computation. Each has a unique name and a body consisting of some Nomadic Pict concurrent process P (modelling whatever multi-threaded programs are running in that virtual machine); at any moment it is located at a particular site. For simplicity we do not model nested virtual machines.

A virtual machine can *relocate*, at any point in time, to any other site (identified by name), new virtual machines can be *created* (with the system synthesising a new unique name, bound to a lexically

P	$::=$	$\texttt{\textbf{create}}^Z\ a = P\ \texttt{\textbf{in}}\ Q$		spawn VM a with body P, on local site
	$\mid$	$\texttt{\textbf{relocate to}}\ s \to P$		relocate this VM to site s
	$\mid$	$\texttt{\textbf{iflocal}}\ \langle a\rangle c\texttt{!}v\ \texttt{\textbf{then}}\ P\ \texttt{\textbf{else}}\ Q$		send $c\texttt{!}v$ to VM a if it is co-located here, and run P, otherwise run Q
	$\mid$	$\langle a\rangle c\texttt{!}v$	(sugar)	send $c\texttt{!}v$ to VM a if it is co-located here
	$\mid$	$\langle a@s\rangle c\texttt{!}v$	(sugar)	send $c\texttt{!}v$ to VM a if it is at site s
	$\mid$	$\mathbf{0}$		empty process
	$\mid$	$P\mid Q$		parallel composition of processes P and Q
	$\mid$	$\texttt{\textbf{new}}\ c : \texttt{\^{}}^I T\ \texttt{\textbf{in}}\ P$		declare a new channel c
	$\mid$	$c\texttt{!}v$		output of v on channel c in current VM
	$\mid$	$c\texttt{?}p \to P$		input on channel c in current VM
	$\mid$	$\texttt{*}c\texttt{?}p \to P$		replicated input
	$\mid$	$\texttt{\textbf{if}}\ v\ \texttt{\textbf{then}}\ P\ \texttt{\textbf{else}}\ Q$		conditional
	$\mid$	$\texttt{\textbf{let}}\ p = ev\ \texttt{\textbf{in}}\ P$		local declaration

Figure 1. Nomadic π-calculus: Syntax

$\Gamma \Vdash @_a\texttt{\textbf{create}}^Z\ b = P\ \texttt{\textbf{in}}\ Q$	$\to$	$\Gamma \Vdash \texttt{\textbf{new}}\ b : \texttt{VM}^Z@s\ \texttt{\textbf{in}}\ (@_b P \mid @_a Q)$	if $\Gamma \vdash a@s$
$\Gamma \Vdash @_a\texttt{\textbf{relocate to}}\ s \to P$	$\to$	$(\Gamma \oplus a \mapsto s) \Vdash @_a P$	
$\Gamma \Vdash @_a\ (c\texttt{!}v \mid c\texttt{?}p \to P)$	$\to$	$\Gamma \Vdash @_a \mathsf{match}(p, v) P$	
$\Gamma \Vdash @_a\texttt{\textbf{iflocal}}\ \langle b\rangle c\texttt{!}v\ \texttt{\textbf{then}}\ P\ \texttt{\textbf{else}}\ Q$	$\to$	$\Gamma \Vdash @_a P \mid @_b c\texttt{!}v$	if $\Gamma \vdash a@s \wedge \Gamma \vdash b@s$
$\Gamma \Vdash @_a\texttt{\textbf{iflocal}}\ \langle b\rangle c\texttt{!}v\ \texttt{\textbf{then}}\ P\ \texttt{\textbf{else}}\ Q$	$\to$	$\Gamma \Vdash @_a Q$	if $\Gamma \vdash a@s \wedge \Gamma \vdash b@s' \wedge s \neq s'$

Figure 2. Nomadic π-calculus: Selected Reduction Rules

scoped identifier) and virtual machines can *interact* by sending messages to each other.

A key point in the design of the low-level calculus is to make it easy to understand the behaviour of the system in the presence of partial failure. To do so, we chose interaction primitives that can be directly implemented above the real-world network (the Sockets API and TCP or UDP), without requiring a sophisticated distributed infrastructure. Our guiding principle is that each reduction step of the low-level calculus should be implementable using at most one inter-site asynchronous communication. [1]

To provide an expressive language for local computation within each virtual machine body, but keep the calculus concise, we include the constructs of a standard asynchronous π-calculus. The Nomadic Pict concurrent process of a virtual machine body can involve parallel composition, new channel creation, and asynchronous messaging on those channels within the virtual machine.

In the rest of this section we give the syntax of processes, and the key points of their reduction semantics.

2.1 Processes of the Low-Level Calculus

The syntax of a low-level core calculus is given in Fig. 1, grouped into the three virtual machine primitives, two useful communication forms that are expressible as syntactic sugar, and the local asynchronous pi-calculus. Executing the construct $\texttt{\textbf{create}}^Z\ b = P\ \texttt{\textbf{in}}\ Q$ spawns a new virtual machine, with body P, on the current site. After the creation, Q commences execution, in parallel with the rest of of the body of the spawning virtual machine. The new virtual machine has a unique name which may be referred to with b, both in its body and in the spawning virtual machine (b is binding in P and Q). The Z is a mobility capability, either `s`, requiring this virtual machine to be static, or `m`, allowing it to be mobile.

Virtual machines can relocate to named sites: the execution of $\texttt{\textbf{relocate to}}\ s \to P$ as part of a virtual machine results in the whole of that virtual machine migrating to site s. After the migration, P commences execution in parallel with the rest of the body of the virtual machine.

There is a single primitive for interaction between virtual machines, allowing an atomic delivery of an asynchronous message between two virtual machines that are co-located on the same site. The execution of $\texttt{\textbf{iflocal}}\ \langle a\rangle c\texttt{!}v\ \texttt{\textbf{then}}\ P\ \texttt{\textbf{else}}\ Q$ in the body of virtual machine b has two possible outcomes. If the virtual machine a is on the same site as virtual machine b then the message $c\texttt{!}v$ will be delivered to a (where it may later interact with an input) and P will commence execution in parallel with the rest of the body of b; otherwise the message will not be delivered and Q will execute as part of b. This is analogous to test-and-set operations in shared memory systems—delivering the message and starting P, or discarding it and starting Q, atomically. It can greatly simplify algorithms that involve communication with virtual machines that may relocate away at any time, yet is still implementable locally, by the VM implementation on a single site.

Two other useful constructs can be expressed as sugar: $\langle a\rangle c\texttt{!}v$ and $\langle a@s\rangle c\texttt{!}v$ attempt to deliver $c\texttt{!}v$ (an output of v on channel c), to virtual machine a, on the current site and on s, respectively. They fail silently if a is not where it is expected to be, and so are usually used only in a context where a is predictable.The first is implementable simply as $\texttt{\textbf{iflocal}}\ \langle a\rangle c\texttt{!}v\ \texttt{\textbf{then}}\ \mathbf{0}\ \texttt{\textbf{else}}\ \mathbf{0}$; the second as $\texttt{\textbf{create}}^{\texttt{m}}\ b = \texttt{\textbf{relocate to}}\ s \to \langle a\rangle c\texttt{!}v \texttt{\textbf{in}}\ \mathbf{0}$, for a fresh name b that does not occur in s, a, c, or v.

[1] This choice may not be appropriate in the virtual machine setting, where one would presumably like to relocate VMs while retaining whatever network connections and connectivity they possess.

Turning to the π-calculus constructs, the body of a virtual machine may be empty (**0**) or a parallel composition $P|Q$ of processes.

Execution of **new** $c : \texttt{\^{}}^I T$ **in** P creates a new unique channel name for carrying values of type T; c is binding in P. The I is a capability: as in (Pierce and Sangiorgi 1996), channels can be used for input only `r`, output only `w`, or both `rw`; these induce a subtype order.

An output $c\texttt{!}v$ (of value v on channel c) and an input $c\texttt{?}p \rightarrow P$ in the same virtual machine may synchronise, resulting in P with the appropriate parts of the value v bound to the formal parameters in the pattern p. Note that, as in other asynchronous π-calculi, outputs do not have continuation processes. A replicated input $\texttt{*}c\texttt{?}p \rightarrow P$ behaves similarly except that it persists after the synchronisation, and so might receive another value.

Finally, we have conditionals **if** v **then** P **else** Q, and local declarations **let** $p = ev$ **in** P, assigning the result of evaluating a simple value expression ev to a pattern p. In $c\texttt{?}p \rightarrow P$, $\texttt{*}c\texttt{?}p \rightarrow P$ and **let** $p = ev$ **in** P the names in pattern p are binding in P.

For a simple example program in the low-level calculus, consider the following VM server.

```
*getVM?[a s] →
  create^m b =
    relocate to s →
      (⟨a@s′⟩ack!b | B)
  in 0
```

It can receive (on the channel named $getVM$) requests for an virtual machine. This is a replicated input ($\texttt{*}getVM\texttt{?}[a\ s] \rightarrow \ldots$) so the server persists and can repeatedly grant requests. The requests contain a pair (bound to the tuple $[a\ s]$ of a and s) consisting of the name of the requesting virtual machine and the name of the site for the new VM to go to. When a request is received the server creates a virtual machine with a new name bound to b. This virtual machine immediately relocates to site s. It then sends an acknowledgement to the requesting virtual machine a (which here is assumed to be on site s') containing its name. In parallel, the body B of the served VM commences execution.

2.2 Processes of the High-Level Calculus

The high-level calculus is obtained by extending the low-level language with a single location-independent communication primitive.

P	$::=$	$\ldots$	
	$\mid$	$\langle a@?\rangle c\texttt{!}v$	send $c\texttt{!}v$ to virtual machine a whereever it is

The intended semantics is that this will reliably deliver the message $c\texttt{!}v$ to virtual machine a, irrespective of the current site of a and of any relocations. The high-level calculus includes all the low-level constructs, so those low-level communication primitives are also available for interaction with application virtual machines whose locations are predictable.

2.3 Outline of the Reduction Semantics

2.3.1 Located Processes and Located Type Contexts

The basic process terms given above only allow the source code of the body of a single virtual machine to be expressed. During computation, this virtual machine may evolve into a system of many virtual machines, distributed over many sites. To denote such systems, we define *located processes*

$$LP ::= @_a P \mid LP|LQ \mid \textbf{new}\ x : T@s\ \textbf{in}\ LP$$

Here the body of a virtual machine a may be split into many parts, for example written $@_a P_1 | \ldots | @_a P_n$. The construct **new** $x : T@s$ **in** LP declares a new name x (binding in LP); if this is a virtual machine name, with $T = \texttt{VM}^Z$, we have an annotation $@s$ giving the name s of the site where the virtual machine is currently located. Channels, on the other hand, are not located – if $T = \texttt{\^{}}^I T'$ then the annotation is omitted.

Correspondingly, we add location information to type contexts. *Located type contexts* Γ include data specifying the site where each declared virtual machine is located; the operational semantics updates this when virtual machines move.

$$\Gamma ::= \bullet \mid \Gamma, X \mid \Gamma, x : \texttt{VM}^Z@s \mid \Gamma, x : T \quad T \neq \texttt{VM}^Z$$

For example, the located type context below declares two sites, s and s', and a channel c, which can be used for sending or receiving integers. It also declares a mobile virtual machine a, located at s, and a static virtual machine b, located at s'.

$$s : \texttt{Site},\ s' : \texttt{Site},\ c : \texttt{\^{}}^{\texttt{rw}}\texttt{Int},\ a : \texttt{VM}^{\texttt{m}}@s,\ b : \texttt{VM}^{\texttt{s}}@s'$$

2.3.2 Reductions

To capture our informal understanding of the calculus in as lightweight a way as possible, we give a reduction semantics. It is defined with a structural congruence and reduction axioms, extending that for the π-calculus (Milner 1993). Reductions are over *configurations*, which are pairs $\Gamma \Vdash LP$ of a located type context Γ and a located process LP. We use a judgement $\Gamma \vdash a@s$, meaning that a virtual machine a is located at s in the located type context Γ. We shall give some examples of reductions, illustrating the new primitives. The most interesting axioms for the low-level calculus are given in Figure 2.

A virtual machine a can spawn a new virtual machine b, with body P, and continues with Q. The new virtual machine is located at the same site as a (say s, with $\Gamma \vdash a@s$). The virtual machine b is initially bound and the scope is over the process Q in a and the whole of the new virtual machine.

$$\begin{array}{ll} & \Gamma \Vdash @_a(R \mid \textbf{create}^{\texttt{m}}\ b = P\ \textbf{in}\ Q) \\ \rightarrow & \Gamma \Vdash @_a R \mid \textbf{new}\ b : \texttt{VM}^{\texttt{m}}@s\ \textbf{in}\ (@_a Q \mid @_b P) \end{array}$$

When a virtual machine a relocates to a new site s, we simply update the located type context.

$$\begin{array}{ll} & \Gamma \Vdash @_a(R \mid \textbf{relocate to}\ s \rightarrow Q) \\ \rightarrow & \Gamma \oplus a \mapsto s \Vdash @_a(R \mid Q) \end{array}$$

A **new**-bound virtual machine may also relocate; in this case, we simply update the location annotation.

$$\begin{array}{ll} & \Gamma \Vdash @_a R \mid \textbf{new}\ b : \texttt{VM}^{\texttt{m}}@s'\ \textbf{in}\ @_b \textbf{relocate to}\ s \rightarrow Q \\ \rightarrow & \Gamma \Vdash @_a R \mid \textbf{new}\ b : \texttt{VM}^{\texttt{m}}@s'\ \textbf{in}\ @_b Q \end{array}$$

A virtual machine a may send a location-dependent message to a virtual machine b if they are on the same site. The message, once delivered may then react with an input in b. Assuming that $\Gamma \vdash a@s$ and $\Gamma \vdash b@s$.

$$\begin{array}{ll} & \Gamma \Vdash @_a(\textbf{iflocal}\ \langle b\rangle c\texttt{!}[]\ \textbf{then}\ P\ \textbf{else}\ Q) \mid @_b(c\texttt{?}[] \rightarrow R) \\ \rightarrow & \Gamma \Vdash @_a P \mid @_b(c\texttt{!}[] \mid c\texttt{?}[] \rightarrow R) \\ \rightarrow & \Gamma \Vdash @_a P \mid @_b R \end{array}$$

If a and b are at different sites then the message will get lost.

$$\begin{array}{ll} & \Gamma \Vdash @_a(\textbf{iflocal}\ \langle b\rangle c\texttt{!}[]\ \textbf{then}\ P\ \textbf{else}\ Q) \mid @_b(c\texttt{?}[] \rightarrow R) \\ \rightarrow & \Gamma \Vdash @_a Q \mid @_b(c\texttt{?}[] \rightarrow R) \end{array}$$

Synchronisation of a local output $c\texttt{!}v$ and an input $c\texttt{?}x \rightarrow P$ only occurs within a virtual machine, but in the execution of **iflocal** a new channel name can escape the virtual machine where it was created, to be used elsewhere for output and/or input. Consider for example the process below, executing as the body of a virtual machine a.

$$\begin{array}{l}
\mathbf{create}^{\mathtt{m}}\ b = \\
\quad c\mathbf{?}x\!\rightarrow (x\mathbf{!}3|x\mathbf{?}n\!\rightarrow\!\mathbf{0}) \\
\mathbf{in} \\
\quad \mathbf{new}\ d : \mathtt{\hat{}}^{\mathtt{rw}}\mathtt{int}\ \mathbf{in} \\
\qquad \mathbf{iflocal}\ \langle b\rangle c\mathbf{!}d\ \mathbf{then}\ \mathbf{0}\ \mathbf{else}\ \mathbf{0} \\
\qquad |\ d\mathbf{!}7
\end{array}$$

It has a reduction for the creation of virtual machine b, a reduction for the **iflocal** that delivers the output $c\mathbf{!}d$ to b, and then a local synchronisation of this output with the input on c. Virtual machine a then has body $d\mathbf{!}7$ and virtual machine b has body $d\mathbf{!}3|d\mathbf{?}n\!\rightarrow\!\mathbf{0}$. Only the latter output on d can synchronise with b's input $d\mathbf{?}n\!\rightarrow\!\mathbf{0}$. For each channel name there is therefore effectively a π-calculus-style channel in each virtual machine. The channels are distinct, in that outputs and inputs can only interact if they are in the same virtual machine. This provides a limited form of dynamic binding, with the semantics of a channel name (i.e., the set of partners that a communication on that channel might synchronise with) dependent on the virtual machine in which it is used; it proves very useful in the infrastructure algorithms that we develop.

The high-level calculus has one additional axiom, allowing location-independent communication between virtual machines.

$$\Gamma \Vdash @_a\langle b@?\rangle c\mathbf{!}v \ \rightarrow\ \Gamma \Vdash @_b c\mathbf{!}v$$

This delivers the message $c\mathbf{!}v$ to virtual machine b irrespective of where b (and the sender a) are located. For example, below an empty tuple message on channel c is delivered to a virtual machine b with a waiting input on c.

$$\begin{array}{rl}
& \Gamma \Vdash @_a(P \mid \langle b@?\rangle c\mathbf{!}[]) \mid @_b(c\mathbf{?}[]\rightarrow R) \\
\rightarrow & \Gamma \Vdash @_a P \mid @_b(c\mathbf{!}[] \mid c\mathbf{?}[]\rightarrow R)
\end{array}$$

3. The Questions

So, are these calculi (or something similar) a level of abstraction that would be useful in managing datacentres, with widespread virtualization? Are there system design problems whose solutions would be best expressed at this level?

Insofar as there are problems involving the interaction of VM relocation and inter-VM communication, the answer seems (plausibly, to us) yes, but we are not in a position to know. We look forward to finding out.

References

1996. *CONCUR '96: Concurrency Theory, 7th International Conference*. LNCS, vol. 1119. Springer-Verlag, Pisa, Italy.

CARDELLI, L. 1995. A language with distributed scope. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*, ACM, Ed. ACM Press, New York, NY, USA, 286–297.

FOURNET, C., GONTHIER, G., LÉVY, J.-J., MARANGET, L., AND RÉMY, D. 1996. A calculus of mobile agents. See (CON 1996), 406–421.

MILNER, R. 1993. The polyadic π-calculus: A tutorial. Series F: Computer and System Sciences, vol. 94. Springer. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

MILNER, R., PARROW, J., AND WALKER, D. 1992. A calculus of mobile processes, part I/II. *Information and Computation 100*, 1–77.

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. The MIT Press.

PIERCE, B. C. AND SANGIORGI, D. 1996. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science 6*, 5, 409–454. An extract appeared in *Proc. LICS '93*: 376–385.

SEWELL, P., WOJCIECHOWSKI, P. T., AND PIERCE, B. C. 1998. Location independence for mobile agents. In *Proceedings of IFL 98: the Workshop on Internet Programming Languages (Chicago), in conjunction with ICCL*. 6pp.

SEWELL, P., WOJCIECHOWSKI, P. T., AND PIERCE, B. C. 1999. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*. Springer Verlag, 1–31.

THOMSEN, B., LETH, L., AND KUO, T.-M. 1996. A Facile tutorial. See (CON 1996), 278–298.

UNYAPOTH, A. AND SEWELL, P. 2001. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proceedings of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (London)*. 116–127.

WOJCIECHOWSKI, P. T. 2001. Algorithms for location-independent communication between mobile agents. In *Proceedings of AISB '01 Symposium on Software Mobility and Adaptive Behaviour (York, UK)*. Also published as Technical Report IC-2001-13, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL).

WOJCIECHOWSKI, P. T. 2006. Scalable message routing for mobile software assistants. In *Proceedings of EUC '06: the 2006 IFIP International Conferenc e on Embedded And Ubiquitous Computing*. Lecture Notes in Computer Science, vol. 4096. Springer, 355–364.

WOJCIECHOWSKI, P. T. AND SEWELL, P. 1999. Nomadic Pict: Language and infrastructure design for mobile agents. In *Proceedings of ASA/MA '99 (First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents), Palm Springs, CA, USA*.

WOJCIECHOWSKI, P. T. AND SEWELL, P. 2000. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency 8*, 2 (April–June), 42–52.