Abstractions for Mobile Computation

Luca Cardelli

August 1, 1998

Technical Report
MSR-TR-98-34

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Abstractions for Mobile Computation

Luca Cardelli

Microsoft Research, 1 Guildhall Street
Cambridge, CB2 3NH, UK
www.luca.demon.co.uk

Abstract. We discuss the difficulties caused by mobile computing and mobile computation over wide-area networks. We propose a unified framework for overcoming such difficulties.

1 Introduction

The Internet and the World-Wide-Web provide a computational infrastructure that spans the planet. It is appealing to imagine writing programs that exploit this global infrastructure. Unfortunately, the Web violates many familiar assumptions about the behavior of distributed systems, and demands novel and specialized programming techniques. In particular, three phenomena that remain largely hidden in local-area network architectures become readily observable on the Web:

- ***(A) Virtual locations***. Barriers are erected between mutually distrustful administrative domains. Therefore, a program must be aware of where it is, and of how to move or communicate between different domains. The existence of separate administrative domains induces a notion of virtual locations and of virtual distance between locations.
- ***(B) Physical locations***. On a planet-size structure, the speed of light becomes tangible. For example, a procedure call to the antipodes requires at least 1/10 of a second, independently of future improvements in networking technology. This absolute lower bound to latency induces a notion of physical locations and physical distance between locations.
- ***(C) Bandwidth fluctuations***. A global network is susceptible to unpredictable congestion and partitioning, which results in fluctuations or temporary interruptions of bandwidth. Moreover, mobile devices may perceive bandwidth changes as a consequence of physical movement. Programs need to be able to observe and react to these fluctuations.

These features may interact among themselves. For example, bandwidth fluctuations may be related to physical location because of different patterns of day and night network utilization, and to virtual location because of authentication and encryption across domain boundaries. Virtual and physical locations are often related, but need not coincide.

In addition, another phenomenon becomes unobservable on the Web:

- ***(D) Failures***. On the Web, there is no practical upper bound to communication delays. In particular, failures become indistinguishable from long delays, and thus undetectable. Failure recovery becomes indistinguishable from intermittent connectivity. Furthermore, delays (and, implicitly, failures) are frequent and unpredictable.

These four phenomena determine the set of *observables* of the Web: the events or states that can be in principle detected. Observables, in turn, influence the basic building blocks of computation. In moving from local-area networks to wide-area networks, the set of observables changes, and so does the computational model, the programming constructs, and the kind of programs one can write. The question of how to "program the Web" reduces to the question of how to program with the new set of observables provided by the Web.

At least one general technique has emerged to cope with the observables characteristic of a wide-area network such as the Web. *Mobile computation* is the notion that running programs need not be forever tied to a single network node. Mobile computation can deal in original ways with the following phenomena:

- ***(A) Virtual locations***. Given adequate trust mechanisms, mobile computations can cross barriers and move between virtual locations. Barriers are designed to impede access, but when code is allowed to cross them, it can access local resources without the impediment of the barrier.
- ***(B) Physical locations***. Mobile computations can move between physical locations, turning remote calls into local calls, and thus avoiding the latency limit.
- ***(C) Bandwidth fluctuations***. Mobile computations can react to bandwidth fluctuations, either by moving to a better-placed location, or by transferring code that establishes a customized protocol over a connection.
- ***(D) Failures***. Mobile computations can run away from anticipated failures, and can move around presumed failures.

Mobile computation is also strongly related to recent hardware advances, since computations move implicitly when carried on portable devices. In this sense, we cannot avoid the issues raised by mobile computation: more than an avant-garde software technique, it is an existing hardware reality.

In this paper, we discuss mobile computation at an entirely informal level; formal accounts of our framework can be found in [10]. In section 2 we describe the basic characteristics of our existing computational infrastructure, and the difficulties that must be overcome to use it effectively. In section 3 we review existing ways of modeling distribution and mobility. In section 4 we introduce an abstract model, the *ambient calculus*, that attempts to capture fundamental features of distribution and mobility in a simple framework. In section 5, we discuss applications of this model to programming issues, including a detailed example and a programming challenge.

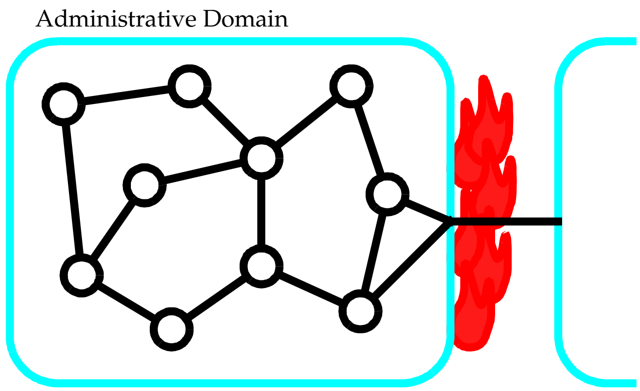
2 Three Mental Images

We begin by comparing and contrasting three *mental images*; that is, three abstracted views of distributed computation. From the differences between these mental images we derive the need for new approaches to global computation.

2.1 Local Area Networks

The first mental image corresponds to the now standard, and quickly becoming obsolete, model of computation over local area networks.

When workstations and PCs started replacing mainframes, local networks were invented to connect autonomous computers for the purpose of resource sharing. A typical local area network consists of a collection of computers of about the same power (within a couple of hardware generations) and of network links of about the same bandwidth and latency. This environment is not always completely uniform: specialized machines may operate as servers or as engineering workstations, and specialized subnetworks may offer optimized services. Still, by and large, the structure of a LAN can be depicted as the uniform network of nodes (computers) and links (connections) in Mental Image 1:



Mental Image 1: Local Area Network

A main property of such a network is its predictability. Communication delays are bounded, and processor response times can be estimated. Therefore, link and process failures can be detected by time-outs and by "pinging" nodes.

Another important property of local area networks is that they are usually well-administered and, in recent times, protected against attack. Network administrators have the task of keeping the network running and protect it against infiltration. In the picture, the boundary line represents an *administration domain*, and the flames represent the protection provided by a *firewall*. Protection is necessary because local area networks are rarely completely disconnected: they usually have slower links to the outside world, which are however enough to make administrators nervous about infiltration.

The architecture of local area networks is very different from the older, highly centralized, mainframe architecture. This difference, and the difficulties implied by it, resulted in the emergence of novel distributed computing techniques, such as remote-procedure-call, client-server architecture, and distributed object-oriented programming. The combined aim and effect of these techniques is to make the programming and application environment stable and uniform (as in mainframes). In particular, the network topology is carefully hidden so that any two computers can be considered as lying one logical step apart. Moreover, computers can be considered immobile; for example, they usually preserve their network address when physically moved.

Even in this relatively static environment, the notion of mobility has gradually acquired prominence, in a variety of forms. For example:

- ***Control mobility***. During an RPC (Remote Procedure Call) or RMI (Remote Method Invocation) call, a thread of control is thought of as moving from one machine to another and back.
- ***Data mobility***. In RPC/RMI, data is linearized, transported, and reconstructed across machines.
- ***Link mobility***. The end-points of network channels, or remote object proxies, can be transmitted.
- ***Object mobility***. For load balancing purposes, objects can be moved between different servers.
- ***Remote Execution***. Computations can be shipped for execution on a server. (This is an early version of code mobility, proposed as an extension of RPC.)

In recent years, distributed computing has been endowed with greater mobility properties and easier network programming. Techniques such as Object Request Brokers have emerged to abstract over the location of objects providing certain services. Code mobility has emerged in Tcl and other scripting languages to control network applications. Agent mobility has been pioneered in Telescript [30], aimed towards a uniform (although wide-area) network of services. Closure mobility (the mobility of active and connected entities) has been investigated in Obliq [9].

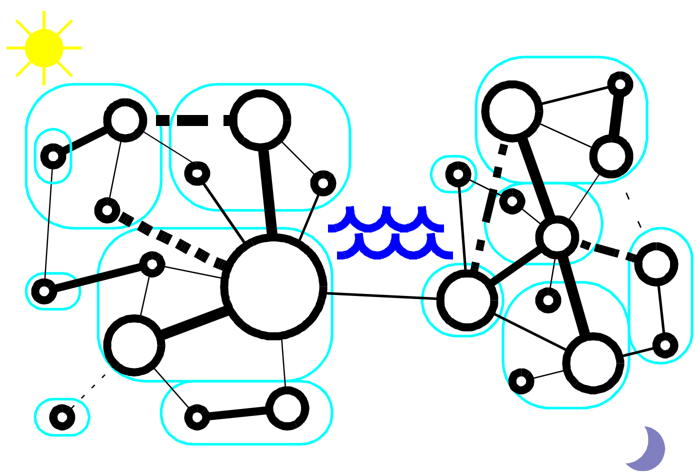
In due time, local-area-network techniques would have smoothly and gradually evolved towards deployment on wide-area networks, e.g. as was explicitly attempted by the CORBA effort. But, suddenly, a particular wide-area network came along that radically changed the fundamental assumptions of distributed computing and its pace of progress: the Web.

2.2 Wide Area Networks

Global computing evolved over the span of a few decades in the form of the Internet. But it was not until the emergence of the Web that the peculiar characteristics of the Internet were exposed in a way that anybody could verify with just a few mouse clicks. For clarity and simplicity we will refer to the Web as the primary global information infrastructure, although it was certainly not the first one.

We should remember that the notions of a *global address space* and of a *global file system* have been popular at times as an extension of mainframe architecture to wide-area networks. The first obvious feature of the Web is that, although it forms a global computational resource, it is nothing like a global mainframe. The Web has no single reliable component, but it also has no single failure point; it is definitely not the centralized all-powerful mainframe of 1950's science fiction novels that could be shut off by attacking its single "brain"[1]. Fortunately, the fact that the Web is not a mainframe is not a big concern; we have already successfully tackled distributed computing based on LANs.

More distressing is the fact that the Web does not behave like a LAN either. Many proposals have emerged along the lines of extending LAN concepts to a global environment; that is, in turning the Internet into a *distributed address space*, or a *distributed file system*. However, since the global environment does not have the stability properties of a LAN, this can be achieved only by introducing redundancy (for reliability), replication (for quality of service), and scalability (for management) at many different levels. Things might have evolved in this direction, but this is not the way the Web came to be. The Web is, almost by definition, unreliable, unpredictable, and unmanageable as a whole, and was not designed with LAN-like guarantees of service.



Mental Image 2: Wide Area Network (for example, the Web)

Therefore, the main problem with the Web is that it is not just a big LAN, otherwise, modulo issues of scale, we would already know how to deal with it. There are several ways in which the Web is not a big LAN, and we will describe them shortly. But the fundamental reason is that, unlike a LAN, the Web is not centrally

[1] Still, a single faulty routing configuration file spread over the Internet in July 1997, causing the disappearance of a large number of Internet domains. In this case, the vulnerable "brain" was the collection of Internet routers.

administered. Instead, it is a dynamic collection of countless independent administrative domains, all widely different and mutually distrustful. This is represented in Mental Image 2.

In that picture, computers differ greatly in power and availability, while network links differ greatly in capacity and reliability. Large physical distances have visible effects, and so do time zones. The architecture of a wide-area network is yet again fundamentally different from that of a local area network. Most prominently, the network topology is dynamic and non-trivial. Computers become intrinsically mobile: they require different addressing when physically moved across administrative boundaries. Techniques based on mobility become more important and sometimes essential. For example, mobile Java applets provided the first disciplined mechanism for running code able (and allowed) to systematically penetrate other people's firewalls. Countless projects have emerged in the last few years with the aim of supporting mobile computation over wide areas, and are beginning to be consolidated.

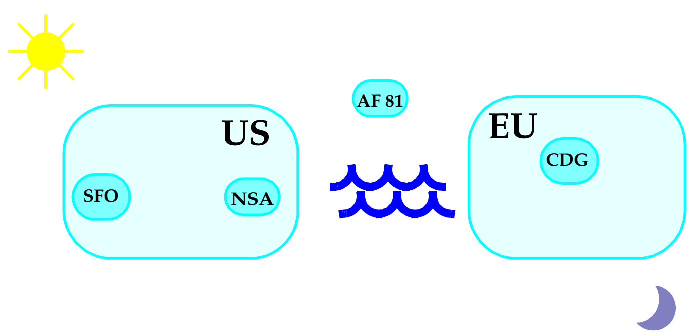
At this point, our architectural goal might be to devise techniques for managing computation over an unreliable collection of far-flung computers. However, this is not yet the full picture. Not only are network links and nodes widely dispersed and unreliable; they are not even liable to stay put, as we discuss next.

2.3 Mobile Computing

A different global computing paradigm has been evolving independently of the Web. Instead of connecting together all the LANs in the world, another way of extending the reach of a LAN is to move individual computers and other gadgets from one LAN to another, dynamically.

We discussed in the Introduction how the main characteristics of the Web point towards mobile comput*ation*. However, that is meant as mobile computation over a fixed (although possibly flaky) network. A more interesting picture emerges when the very components of the network can move about. This is the field of mobile comput*ing*. Today, laptops and personal organizers routinely move about; in the future entire networks will go mobile (as in IBM's Personal Area Network). Existing examples of this kind of mobility include: a smart card entering a network computer slot; an active badge entering a room; a wireless PDA or laptop entering a building; a mobile phone entering a phone cell.

We could draw a picture similar to Mental Image 1, but with mobile devices moving within the confines of a single LAN. This notion of a dynamic LAN is a fairly minor extension of the basic LAN concepts, and presents few conceptual problems (wireless LANs are already common). A much more interesting picture emerges when we think of mobile gadgets over a WAN, because administrative boundaries and multiple access pathways then interact in complex ways, as anybody who travels with a laptop knows all too well.



Mental Image 3: Mobile Computing

Mental Image 3 focuses on two domains: the United States and the European Union, each enclosed by a political boundary that regulates the movement of people and computers. Within a political boundary, private companies and public agencies may further regulate the flow of people and devices across their doors. Over the Atlantic we see a third domain, representing Air France flight 81 travelling from San Francisco to Paris. AF81 is a very active mobile computational environment: it is full of people working with their laptops and possibly connecting to the Internet through airphones. (Not to mention the hundreds of computers that control the airplane and let it communicate with ground stations.)

Abstracting a bit from people and computation devices, we see here a hierarchy of boundaries that enforce controls and require permissions for crossing. Passports are required to cross political boundaries, reservations are required for restaurants, tickets are required for airplanes, and special clearances are required to enter (and exit!) agencies such as the NSA. Sometimes, whole mobile boundaries cross in and out of other boundaries and similarly need permissions, as the mobile environment of AF81 needs permission to enter an airspace. On the other hand, once an entity has been allowed across a boundary, it is fairly free to roam within the confines of the boundary, until another boundary needs to be crossed.

2.4 General Mobility

We have described two different notions of mobility. The first, *mobile computation*, has to do with virtual mobility (mobile software). The second, *mobile computing*, has to do with physical mobility (mobile hardware). These two fields are today almost disconnected, the first dominated by a software community, and the second dominated by a hardware community. However, the borders between virtual and physical mobility are fuzzy, and eventually we will have to treat all kinds of mobility in a uniform way. Here are two examples where the different forms of mobility interact.

The first example is one of virtual mobility achieved by physical means. Consider a software agent in a laptop. The agent can move by propagating over the network, but can also move by being physically transported with the laptop from one location to another. In the first case, the agent may have to undergo security checks when it crosses administration domains. In the second case the agent may have to undergo security checks (e.g. virus detection) when the laptop is physically allowed inside a new administrative domain. Do we need two completely separate security infrastructures for these two cases, or can we somehow find a common principle? A plausible security policy for a given domain would be that a physical barrier (a building door) should provide the same security guarantees as a virtual barrier (a firewall).

The second example is one of physical mobility achieved by virtual means. Software exists that allows remote control of a computer, by bringing the screen of a remote computer on a local screen. The providers of such software may claim that this is just as good as moving the computer physically, e.g. to access its local data. Moreover, if the remote computer has network connection, this is also equivalent to "stringing wire" from the remote location, since the remote network is now locally accessible. For example, using remote control over a phone line to connect from home to work where a high-bandwidth Internet connection is available, is almost as good as having a high-bandwidth Internet connection brought into the home.

In the future we should be prepared to see increased interactions between virtual and physical mobility, and we should develop frameworks where we can discuss and manipulate these interactions.

2.5 Barriers and Action-at-a-Distance

The unifying difficulty in both mobile computing and mobile computation is the proliferation of barriers, and the problems involved in crossing them. This central difficulty implies that we must regard barriers as fundamental features of our computational models. This seems contrary to the usual trend.

Access barriers have arisen many times in the history of computing, and one of the main tasks of computer science has been to "abstract them away", often by the proverbial extra level of indirection. For example, physical memory boundaries are circumvented by virtual memory; address space boundaries are circumvented by network proxies; firewall boundaries are circumvented by secure tunnels and agent sandboxing. Unfortunately, when barriers are not purely technological it is not possible to completely abstract them away. The crossing of administrative barriers must be performed by explicit bureaucratic operations, such as exhibiting equipment removal passes and export licences.

Therefore, administrative barriers constitute a nasty fundamental change in the way we compute. Let's review some historical examples.

In the early days of the Internet, any computer could talk to any other computer by knowing its IP number. We can now forget about flat IP addressing and transparent routing: routers and firewalls effectively hide certain IP addresses from view and make them unreachable by direct means.

In the early days of programming languages, people envisioned a universal address space in which all programs would live and share data, possibly with world-wide garbage-collection, and possibly with strong typing to guarantee the integrity of the pointers. We can now forget about universal addressing: although pointers are allowed across machines on a LAN (by network proxies), they are generally disallowed across firewalls. Similarly, we can forget about transparent distributed object systems: some network objects will be kept well hidden within certain domains, and reaching them will require effort.

In the early days of mobile agents, people envisioned agents moving freely across the network on behalf of their owners. We can now forget about free-roaming: if sites do not trust agents they will not allow them in. If agents do not trust sites to execute them fairly, they will not want to visit them.

In general, we can forget about the notion of *action-at-a-distance computing*: the idea that resources are available transparently at any time, no matter how far away. Instead, we have to get used to the notion that movement and communication are step-by-step activities, and that they are visibly so: the multiple steps involved cannot be hidden, collapsed, or rendered atomic.

The action-at-a-distance paradigm is still prevalent within LANs, and this is another reason why LANs are different from WANs, where such an assumption cannot hold.

2.6 Why a WAN is not a big LAN

We have already discussed in the Introduction how a WAN exhibits a different set of observable than a LAN. But could one emulate a LAN on top of a WAN, restoring a more familiar set of observables, and therefore a more familiar set of programming techniques? If this were possible, we could then go on and program the Internet just like we now program a LAN.

To turn a WAN into a LAN we would have to hide the new observables that a WAN introduces, and we would have to reveal the observables that a WAN hides. These tasks ranges from difficult, to intolerable, to impossible. Referring to the classification in the Introduction, we would have to achieve the following.

(A) Hiding virtual locations. We would have to devise a security infrastructure that makes navigation across multiple administration domains painless and transparent (when legitimate). Although a great deal of cryptographic technology is available, there may be impossibility results lurking in some corners. For example, it is far from clear whether one can in principle guarantee the integrity of mobile computations against hostile or unfair servers [27]. (This can be solved on a LAN by having all computers under physical supervision.)

(B) Hiding physical locations. One cannot "hide" the speed of light; techniques such as caching and replication may help, but they cannot fool processes that attempt to perform long-distance real-time control and interaction. In principle, one could make all delays uniform, so that programs would not behave differently in different places. Ultimately this can be achieved only by slowing down the entire infrastructure, by embedding the maximal propagation delay in all communications. (This would be about 1/10 of a second on the surface, but would grow dramatically as the Web is extended to satellite communication, orbital stations, and further away.)

(C) Hiding bandwidth fluctuations. It is possible to introduce service guarantees in the networking infrastructure, and therefore eliminate bandwidth fluctuations, or reduce them below certain thresholds. However, in overload situations this has the only effect of turning network congestion into access failures, which brings us to the next point.

(D) Revealing failures. We would have to make failures as observable as on a LAN. This is where we run into fundamental trouble. A basic result in distributed systems states that we cannot achieve distributed consensus (such as agreeing on which nodes have failed) in a system consisting of a collection of asynchronous processes [15]. The Web is such a system: we can make no assumption about the relative speed of processors (they may be overloaded, or temporarily disconnected), about the speed of communication (the network may be congested or partitioned), about the order of arrival of messages, or even about the number of processes involved in a computation. In these circumstances, it is impossible to detect the failure of processors or of network nodes or links; any consensus algorithm can be delayed indefinitely. The common partial solutions for this unsolvable problem are to dictate some degree of synchrony and failure detection. These solutions work well on a LAN, but they seem unlikely to apply to WANs simply because individual users may arbitrarily decide to turn off their processors without warning, or take them into unreachable places. Other partial solutions involve multiple-round broadcast-based probabilistic algorithms [7] which might be expensive on a WAN in terms of communication load, and would be subject to light-speed delays. Moreover, it is difficult to talk about the failure of processors that are invisible because they are hidden behind firewalls, and yet take part in computations. Therefore, it seems unlikely that techniques developed to deal with asynchrony in operating systems and LANs can be successfully applied to a WAN such as the Web in full generality. The Web is an inherently asynchronous system, and the impossibility result of [1] applies with full force.

In summary: task (A) may be unsolvable for mobile code; in any case, a non-zero amount of bureaucracy will always be required; task (B) is only solvable (in full) by introducing unacceptable delays; task (C) can be solved in a way that reduces it to (D); task (D) is unsolvable in principle, while probabilistic solutions run into task (B).

2.7 Mobility Postulates

We summarize this section by a collection of postulates that capture the main properties of the reality we are interested in modeling:

- 1) Separate locations exist.
- 2) Since different locations have different properties, both people and programs will want to move between them.
- 3) Barriers to mobility will be erected to preserve certain properties of certain locations.
- 4) Some people and some programs will still need to cross those barriers.

The point of these postulates is to stress that, although we can debate what the "killer applications" for mobility might be, mobility and barrier crossing are *inevitable* requirements of our current and future computing infrastructure.

3 Modeling Mobility

Section 2 was dedicated to showing that the reality of mobile computation over a WAN does not fall into familiar categories. Therefore, we need to invent a new paradigm, or model, that can help us in understanding and eventually in taking advantage of this reality. Since the Web is, after all, a distributed system, it is worth reviewing the existing literature on models of distributed systems to see if there is something there that we can already use.

3.1 Formalisms for Concurrency, Distribution, and Security

The π-calculus [24], along with its variations, is a prominent model of concurrency, and is the starting point for our work. We spend some time discussing why some of the basic assumptions of the π-calculus and of other concurrent formalisms do not satisfy our particular needs.

3.1.1 Static Connectivity

Some early paradigms for concurrency, such as Actors [2], allowed highly dynamic systems. However, the main formalized descriptions of concurrency began by considering only static connectivity. This is the case for Petri Nets [8], for Hoare's Communicating Sequential Processes (CSP) [19], and for Milner's Calculus of Communicating Systems (CCS) [22]. In CCS, in particular, the set of communication channels that a processes has available does not change during execution. In some versions of CSP (and in Occam [21]) the set of processes cannot not change either.

These models are insufficient for modeling mobility in a general sense. Instead, they best characterize the situation in Mental Image 1, where the only form of mobility is the mobility of data over communication channels.

3.1.2 Dynamic Connectivity

The π-calculus is an extension of CCS where channels can be transmitted over other channels, so that a process can dynamically acquire new channels. Channel transmission (also called channel mobility) is a powerful extension of the basic communication model. The transmission of a channel over another channel gives the recipient the ability to communicate on that channel. It is perhaps best to think that a channel end-point has been transmitted.

Let us consider a channel end-point that is transmitted across a domain boundary over another channel that already crosses the boundary. If the transmitted end-point remains functional, it provides a dynamically-established connection between the two sides of the boundary. This is the kind of connection that firewalls typically forbid: opening arbitrary network connections or allowing network-object proxy requests is not allowed without further checks. The new channel that crosses the firewall could be seen as a firewall tunnel, but the establishment of trusted tunnels involves more than simply passing a channel over another one, otherwise the firewall would loose all control of security. In summary, ability to communicate on a channel should depend not only on possessing the end-point of a channel, but also on where the other end-point of the channel is, and how it got there.

Our approach: We provide a framework where processes exists in multiple disjoint locations, and such that the location of a process influences its ability to communicate with other processes. Dynamic connectivity is achieved by movement, but movement does not guarantee continued connectivity.

3.1.3 Distribution

The π-calculus has no inherent notion of distribution. Processes exist in a single contiguous location, by default, because there is no built-in notion of distinct locations and their effects on processes. Interaction between processes is achieved by global, shared names (the channel names); therefore every process is assumed to be within earshot of every other process. Incidentally, this makes distributed implementation of the original π-calculus (and of CCS) quite hard, requiring some form of distributed consensus.

Various proposals have emerged to make the π-calculus suitable for distributed implementation, and to extend it with location-awareness. The asynchronous π-calculus [20] is obtained by a simple weakening of the π-calculus synchronization primitives; this removes the need for distributed consensus during synchronization [26]. The join-calculus [16] approaches the same problem by rooting each channel at a particular process; this provides a single place where synchronization is resolved. LLinda [14] is a formalization of Linda [11] using process calculi techniques. As in distributed versions of Linda, LLinda has multiple distributed tuple spaces, each with its local synchronization manager.

Our approach: We restrict communication to happen within a single location, so that communication can be locally managed. In particular, interaction is by shared location, not by shared names. Remote communication is modeled by a combination of mobility and local communication.

3.1.4 Locality

By locality we mean here distribution-awareness: a process has some notion of the location it occupies, in an absolute or relative sense.

A growing body of literature is concentrating on the idea of adding discrete locations to a process calculus and considering failure of those locations [3, 17]. A notion of locations alone is not sufficient, since locations could all really be in the "same place". However, in presence of observable failures one could observe that certain locations have failed and others have not, and deduce that those locations are truly in different places, otherwise they would all have failed at the same time. The distributed join-calculus [17], for example, adds a notion of named locations, and a notion of distributed failure; locations form a tree, and subtrees can migrate from one part of the tree to another, therefore becoming subject to different observable failure patterns.

This failure-based approach aims to model traditional distributed environments, and traditional algorithms that tolerate node failures. However, on the Internet, node failure is almost irrelevant compared with inability to reach nodes. Web servers do not often fail forever, but they frequently disappear from sight because of network or node overload, and then they come back. Sometimes they come back in a different place, for example, when a Web site changes its Internet Service Provider. Moreover, inability to reach a Web site only implies that a certain path is unavailable; it implies neither failure of that site nor global unreachability. In this sense, a perceived node failure cannot simply be associated with the node itself, but instead is a property of the whole network, a property that changes over time.

Our approach: The notion of locality is induced not by failures, but by the need to cross barriers. Barriers produce a non-trivial and dynamic topology of locations. Locations are observably distinct because it takes effort to move between them, and because some locations may be "absent" and are distinguishable from locations that are "present". Failure is only represented, in a weak but realistic sense, as becoming forever unreachable.

3.1.5 Mobility

There are different senses in which people talk about "process mobility"; we try to distinguish between them.

A π-calculus channel is mobile in the sense that it can be transmitted over another channel. Let's imagine a process as having a mass, and its channels as springs connecting it to other processes. When springs are added and removed by communication, the process mass is pulled in different directions. Therefore, the set of channels of a process influences its location, and a change of channels causes the process to change location. This is particularly clear if a process has a single active channel at any time, because a single spring will strongly influence a process location. By this analogy, channel mobility can be interpreted as causing process mobility.

However, our desired notion of process mobility is tied to the idea of crossing domain barriers. This is a very discrete, on-off, kind of mobility: either a process is inside a domain, or it is not. Representing this kind of mobility by adding and removing channels is not immediate. For example, if a π-calculus channel crosses a barrier (that is, if it is communicated to a process meant to represent a barrier), there is still no clear sense in which the process has also crossed the barrier. In fact, the same channel may cross several disjoint domain barriers, but a process should not exist in all those domains at once. Therefore, a π-calculus process can be made to span several domains, which is something we would like to rule out from the start.

Our notion of process mobility is also tied to the notion of nested domains. It still remains to be seen how simple hierarchies, such as the ones we have emphasized in our mental images, can capture the sometimes complex relationships between administrative domains. Still, some notion of domain nesting seems natural, and this is difficult to represent adequately in process calculi like the π-calculus that are based on "flat" sets of processes.

Another, more direct, form of process mobility has been proposed: a process may move by being transmitted over a channel to another process. This mechanism is not present in the basic π-calculus, but is present in so-called higher-order π-calculi[2]. This is certainly a form of process mobility of the kind found in mobile agent systems, where "whole" and "alive" agents are transmitted over communication channels. However, there are a couple of details that should raise some concern, again having to do with domain boundaries.

[2] The fact that the latter can be formally reduced to the former [28] is best ignored for this discussion.

First, transmitting a process through a firewall over a channel relies on the existence of a channel that already crosses a firewall. That is, we still have to separately model the establishment of firewall tunnels.

Second, if a process has connections before the move, does it maintain them on the other side of the firewall? If the answer is yes, then arbitrary channels can be made to penetrate a firewall from the existence of a single channel. If the answer is no, why and how do those other channels "break"?

Third, transmitting a process over a channel is a "copy" rather than "move" operation. Nothing prevents the original process from continuing, and nothing prevents the transmitted process from being further replicated. That is, the idea that *the same process* moved from here to there seems to be lost.

Our approach: mobility is not represented by passing channels over channels, nor by passing processes over channels. It is represented by processes jumping across boundaries. The identity of the moving process is preserved because a process that crosses a boundary disappears from its previous location.

3.1.6 Security

It is possible to extend a concurrent calculus with cryptographic primitives, as in the spi calculus [1] extension of the π-calculus. Much fundamental progress is being made in this direction.

In our approach, security is not tied to cryptographic primitives, but on the ability or inability to cross barriers, which is conferred by capabilities. Given the mechanisms already required to handle boundaries, the need for cryptographic extensions does not arise immediately. For example, a boundary enclosing a piece of text can be seen as an encryption of the text, in the sense that a capability, the cryptokey, is needed to cross the boundary and read the text. There is an unexpected similarity between a firewall surrounding a major company and the encryption of a piece of data, both being barrier-based security mechanisms at vastly different scales.

4 Ambients

We have now sufficiently discussed our design constraints and the deficiencies of existing solutions, and we are finally ready to explain our own proposal in detail. We want to capture in an abstract way, notions of locality, of mobility, and of ability to cross barriers. To this end, we focus on *mobile computational ambients*; that is, places where computation happens and that are themselves mobile.

4.1 Overview

Briefly, an *ambient*, in the sense in which we are going to use this word, is a place that is delimited by a boundary and where computation happens. Each ambient has a *name*, a collection of local *processes*, and a collection of *subambients*. Ambients can move in an out of other ambients, subject to *capabilities* that are associated with ambient names. Ambient names are unforgeable, this fact being the most basic security property.

In further detail, an ambient has the following main characteristics.

- An ambient is a *bounded* place where computation happens.

  If we want to move computations easily we must be able to determine what parts should move. A boundary determines what is inside and what is outside an ambient, and therefore determines what moves. Examples of ambients, in this sense, are: a web page (bounded by a file), a virtual address space (bounded by an addressing range), a Unix file system (bounded within a physical volume), a single data object (bounded by "self") and a laptop (bounded by its case and data ports). Non-examples are: threads (where the boundary of what is "reachable" is difficult to determine) and logically related collections of objects. We can already see that a boundary implies some flexible addressing scheme that can denote entities across the boundary; examples are symbolic links, URLs (Uniform Resource Locators) and Remote Procedure Call proxies. Flexible addressing is what enables, or at least facilitates, mobility. It is also, of course, a cause of problems when the addressing links are "broken".

- Ambients can be nested within other ambients, forming a tree structure.

As we discussed, administrative domains are (often) organized hierarchically. Mobility is represented as navigation across a hierarchy of ambients. For example, if we want to move a running application from work to home, the application must be removed from an enclosing (work) ambient and inserted in a different enclosing (home) ambient.

- Each ambient has a collection of local running processes;

  A local process of an ambient is one that is contained in the ambient but not in any of its subambients. These "top level" local processes have direct control of the ambient, and in particular they can instruct the ambient to move. In contrast, the local processes of a subambient have no direct control on the parent ambient: this helps guaranteeing the integrity of the parent.

- Each ambient moves as a whole with all its subcomponents.

  The activity of a single local process may, by causing movement of its parent, influence the location, and therefore the activity, of other local processes and subambients. For example, if we move a laptop and reconnect it to a different network, then all the threads, address spaces, and file systems within it move accordingly and automatically, and have to cope with their new surrounding. Agent mobility is a special case of ambient mobility, since agents are usually single-threaded. Agents, like ambients, automatically carry with them a collection of private data as they move.

- Each ambient has a name.

  The name of an ambient is used to control access (entry, exit, communication, etc.). In a realistic situation the true name of an ambient would be guarded very closely, and only specific capabilities based on the name would be handed out. In our examples we are usually more liberal in the handling of names, for the sake of simplicity.

We have developed and are still studying a formal calculus of ambients [10]. In this paper we stay away from formalism; the essential features of the ambient calculus will be conveyed by (fairly precise) metaphors.

As a first metaphor, already partially outlined, we can consider a foreign travel scenario. A country with border guards can be seen as an example of a static ambient. Local officers (privileged processes) govern movement across the border and communication within the border. Entry and exit capabilities (passports and visas) moderate mobility across the border. Officers may decide who may join them as new officers.

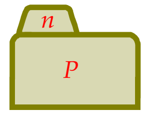
As an example of a mobile ambient we may consider a tourist traveling to such a country. The tourist travels by being transported by another mobile ambient: a train, or airplane. The tourist's visas are checked on entry and exit. The tourist may be imprisoned within the country (by withholding the exit capability) or, if not careful, even killed. On the other hand, the tourist may be allowed to become a local officer (be naturalized), and in such a function may act as a spy or a saboteur, sending out encrypted messages or disabling subsystems.

In the following section we discuss a more abstract metaphor that is easy to describe and has an intuitive graphical presentation. It corresponds faithfully to the full, formal, ambient calculus.

4.2 The Folder Calculus: Mobility

In this office-style metaphor we represent an ambient as a folder. A folder confines its contents: something is either inside or outside any given folder. Each folder has a name that is written on the folder tag. Folders are naturally nested, and can be moved from place to place.

The computational aspect of the calculus is represented by assuming that folders are active. In addition to subfolders, folders may contain *gremlins* that cause the folder to move around, and are moved together with their folder.



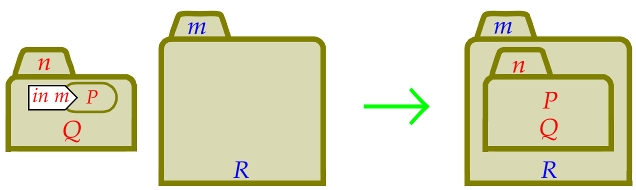
A folder with label n and contents P

We now describe the *reductions* (atomic computation steps) that can arise in the folder calculus. Each reduction can happen at the "top level" of the computation (say, "on the desktop") or inside a folder. That is, the folder hierarchy is active at every level, with some caveats noted below.

4.2.1 The Enter Reduction

Our first reduction causes a folder to enter another folder. On the left of the arrow, in the figure below, the folder labeled n contains a gremlin that is ready to execute the instruction *in* m (meaning: "let the surrounding folder enter a folder labeled m"), and then continue with P. (The partial occlusion of P in the figure is meant to indicate that the instruction *in* m must execute before P can be activated.) The collection of other gremlins and subfolders that may be contained in n is represented by the letter Q, and similarly by R within m. A folder labeled m happens to exist near the folder n, where "near" means "within the same surrounding folder, or on the desktop".

In this situation the operation *in* m can execute, resulting in the configuration to the right of the arrow. The result of the operation is that the folder n becomes a subfolder of the folder m, and the gremlin who executed the instruction is ready to continue with P. The instruction *in* m has been consumed. Any other gremlins or subfolders in Q and R are unchanged.



Enter reduction

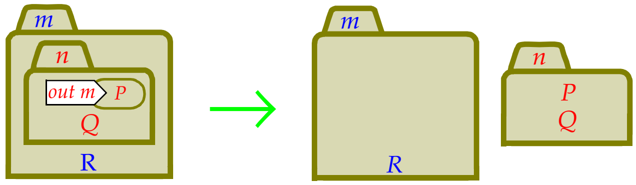
A reduction can happen only if the conditions on the left of the arrow are satisfied. That is, *in* m executes only if there is a sibling folder labeled m. Otherwise, the operation remains blocked until a sibling folder labeled m appears nearby. Such a folder may appear, for example, because it moves near n, or because some of the gremlins in Q cause n to move near it.

Many reductions can be simultaneously enabled, in which case one is chosen nondeterministically. For example, there could be two distinct folders labeled m near n, in which case n could enter either one. Also, there could be another gremlin, as part of Q, trying to execute an *in* m operation, in which case exactly one of the gremlins would succeed. Moreover, there could be another gremlin in Q trying to execute an *in* p operation, and there could be another folder labeled p near n, in which case the n folder could enter either m or p.

4.2.2 The Exit Reduction

Our second reduction is essentially the inverse of the previous one: two nested folders become siblings. On the left of the arrow, the folder labeled n contains a gremlin that is ready to execute the instruction *out* m (meaning: "let the surrounding folder exit its parent labeled m"), and then continue with P. The parent folder is in fact labeled m.

In this situation the operation *out* m can execute, resulting in the configuration to the right of the arrow. The result of the operation is that the folder n becomes a sibling of the folder m, and the gremlin who executed the instruction is ready to continue with P. The instruction *out* m has been consumed. Any other gremlins or subfolders in Q and R are unchanged.



Exit reduction

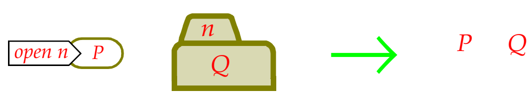
The operation *out* m executes only if the parent folder is labeled m. Otherwise, the operation remains blocked until a parent folder labeled m materializes. The parent may become m, for example, because some of the gremlins in Q cause n to move inside a folder labeled m, at which point *out* m can execute.

Again, several reductions may be enabled at the same time. For example, there could be a gremlin in Q trying to execute an *in* p operation, and there could be a folder labeled p in R. Then, the folder n could either exit m or enter p.

4.2.3 The Open Reduction

Our third reduction is used to discard a folder while keeping its contents. In the picture it is helpful to imagine that there is a folder surrounding the entities on the left of the arrow, so that *open* n followed by P is a gremlin of that folder, and n is one of its subfolders. The gremlin is ready to execute the instruction *open* n (meaning: "let a nearby folder labeled n be opened"), and then continue with P. A folder labeled n happens to exist nearby.

In this situation the operation *open* n can execute, resulting in the configuration to the right of the arrow. The result of the operation is that the folder n is discarded, but its contents Q are spilled in the place where the folder n used to be. The instruction *open* n has been consumed, and the gremlin is ready to continue with P.

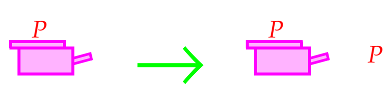


Open reduction

As before, the operation *open* n is blocked if there are no folders labeled n nearby. If there are several ones, any one of them may be opened.

4.2.4 The Copy Reduction

Our fourth reduction is used to replicate folders and all their contents. On the left of the arrow, a copy machine is ready to make copies of a folder P (actually, P could also be a gremlin or any collection of folders and gremlins). We should imagine that the original P is firmly placed under the cover of the copy machine, and is compressed into immobility: none of the gremlins or folders of P can operate or move around (otherwise the copy might be "blurred"). However, as soon as a copy of P is made, that copy is free to execute.



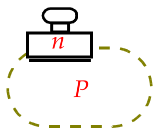
Copy reduction

The copy machine can produce a new copy of P and of all its contents at will (nobody needs to push the copy button). After that, the copy machine can operate again, indefinitely. So, on the right of the arrow we have the same configuration as on the left, plus a fresh copy of P. We could think that copies of P are made on demand, whenever needed, rather than being continuously produced.

4.2.5 *Name Creation*

The handling of names is a delicate and fundamental part of our calculus. Fortunately, it is very well understood: it comes directly from the π-calculus.

Name creation is an implicit operation, in the sense that there are no reductions associated with it. It is represented below as the creation of a rubber stamp for a name n, which can be used to stamp folder labels. Any number of folders can be stamped with the same rubber stamp.



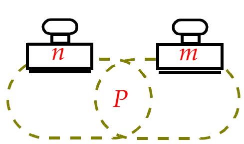
A rubber stamp is used not as much to give a name, as to give *authenticity* to a folder. There are several components to this notion, and we need to stretch our office metaphor a bit to make it fit with the intended semantics.

At a microscopic scale, each rubber stamp has slight imperfections that can be used to tell which rubber stamp was used to stamp each particular folder. Therefore, the particular name chosen for a rubber stamp is irrelevant: what is really important is the relationship between a rubber stamp and the folders it has stamped. Humans, however, like to read names, not microscopic imperfection, so we keep names associated with rubber stamps and folder labels. Still, we are free to change those names at any time, as long as this is done consistently for a rubber stamp, all its related folders, and all the other uses of the name on the rubber stamp. This consistent renaming could be considered as a reduction, but it is simple enough to be considered as a basic equivalence between configurations of folders, expressing the fact that superficial names are not really important.

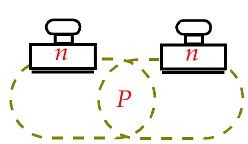
In our metaphor, a copy machine can be used to copy anything contained in a folder, including rubber stamps. Therefore, even if we started with all the rubber stamps having different names, eventually there might be multiple rubber stamps carrying the same name. To make authenticity work, we have to assume that copy machines cannot copy rubber stamps perfectly at the microscopic level: when a rubber stamp is replicated, a different set of microscopic imperfections is generated. That is, rubber stamps are unforgeable by assumption.

For all these reasons, two rubber stamps carrying the same name n are really two different rubber stamps. To preserve authenticity, we do not want these rubber stamps, and the folders they stamp, to get confused. In our visual representation, we collect all the folders stamped by a rubber stamp, and all the other occurrences of its name, within a dashed boundary: this way we can always tell, graphically, which folders were authenticated by a rubber stamp, even when different rubber stamps have the same name.

This dashed border is a flexible boundary and can move about fairly freely (it is just a bookkeeping device). We have three main invariants for where a dashed border can be placed. First it must always be connected with its original rubber stamp. Second, it must always enclose all the folders that have ever been stamped with its particular stamp and all other occurrences of the name (e.g. within gremlin code); if a folder moves away, the dashed boundary may have to be enlarged. Third, dashed boundaries for two rubber stamps with the same name must not intersect; if we should ever need to do so we shall first systematically rename one of the two rubber stamps and the related names, so that there is no confusion. The dashed boundaries for rubber stamps with different names can freely intersect.



allowed



forbidden

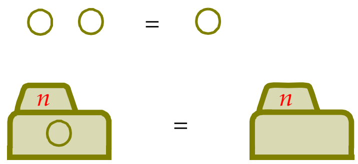
It is allowable to nest dashed boundaries for rubber stamps with the same name: an occurrence of that name will refer to the closest enclosing rubber stamp, in standard block scoping style.

4.2.6 *Leaves of the Syntax*

At the bottom of our syntax there is the inactive gremlin, which can be represented as an empty border. The inactive gremlin has no reductions.

Inactive gremlin

Inactive gremlins are often simply discarded or omitted. For example, multiple inactive gremlins can be collapsed into one inactive gremlin, and a folder containing only one inactive gremlin is usually represented as an empty folder.



Programs in the folder calculus are built from these foundations, by assembling collections of gremlins, folders, rubber stamps, and copy machines, and possibly placing them inside other folders.

4.2.7 *The Theoretical Power of Mobility*

Before moving on to our next and final reduction, we pause and consider the operations we have introduced so far. These operations are purely combinatorial, that is, they introduce no notion of parameter passing, or message exchange. Also, they deal purely with mobility, not with communication or computation of any familiar kind. Yet, they are computationally complete: Turing machines can be encoded in a fairly direct way (see [10]).

Moreover, very informally, it is possible to see an analogy between the Enter reduction and increment, the Exit reduction and decrement, the Open reduction and test, and the Copy reduction and iteration. Therefore all the ingredients for performing arithmetic are present, and it is in fact possible to represent numbers as towers of folders, with appropriate operations.

Data structures such as records and trees can be represented easily, by nested folders; folder names represent pointers into such data structures. Storage cells can be represented as folders whose contents change in response interactions with other folders; in this case a folder name represents the address of a cell.

In summary, the mobility part of the Folder Calculus already has the full power of any computationally complete language. In principle, one could use it as a graphical scripting language for the office/desktop metaphor.

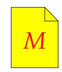
4.3 The Folder Calculus: Communication

Mobility is by itself a computationally complete paradigm. Still, we want to talk about communication in a direct way; for example, gremlins may want to exchange names of folders to visit. Representing communication within the existing set of mobility primitives is theoretically possible, but unbearably clumsy. So, our next task is to introduce communication primitives that do not invalidate the principles described in Section 3, and that do not spoil the folder metaphor. The primitives we introduce next are probably the simplest imaginable. In particular, they do not conflict with the notion of strict folder containment, and do not duplicate mobility functionality.

4.3.1 The Read Reduction

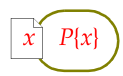
We begin by introducing a new entity that can sit inside a folder: a message. To remain within the office metaphor, we imagine writing messages onto throw-away Post-it notes that are attached to the inside of folders.

A gremlin can write the name of a folder on a note, and can attach the note to the current folder (the folder the gremlin is in). This is an output operation, and is represented graphically by a message written on a note. We shall discuss shortly what are the particular messages used in the folder calculus. More generally, we may imagine writing any kind of data as a message; in this view, a note can be seen as nameless data file that is kept within a folder.



Output

Conversely, a gremlin can grab any note attached to the current folder, read its message, discard the note, and proceed with the knowledge of the message. This is an input operation, and is represented graphically by a process P with occurrences of the variable x (written $P\{x\}$) that is waiting for a note with a message to be bound to x.



Input

The Read reduction is the interaction between input and output operations or, equivalently, between message notes and input operations. In a situation where an input and an output are present, the Read reduction can execute, resulting in the configuration on the right of the arrow, which is simply $P\{M\}$: the residual gremlin P that has read M into x. The note that appears on the left of the arrow is discarded: it is consumed by reading and does not appear on the right. (If the note needs to persist, it can be replicated by a copy machine.)

Read reduction

An input operation blocks if there are no available messages. Several input operations may contend for the same message and only one of them will obtain it and be able to proceed. An output operation, however, never blocks (it is *asynchronous*); it simply drops a message in the current folder and has no continuation.

Inputs and outputs usually happen within a folder (although the reduction above allows them to happen on the desktop). Within such a folder, the identity of the input or output gremlins are not important: anybody can talk to anybody else, in the style of a chat room. In practice, different folders will be dedicated to different kinds of conversation, so one can make assumptions about what should be said and what can be heard in a given folder.

4.3.2 Messages and Capabilities

We can imagine many kinds of messages that can be written on Post-it notes. Here we focus on messages that are *capabilities*: they allow the reader of a message to perform privileged actions. There are two kinds of capabilities, used in different contexts: *naming capabilities* and *navigation capabilities*.

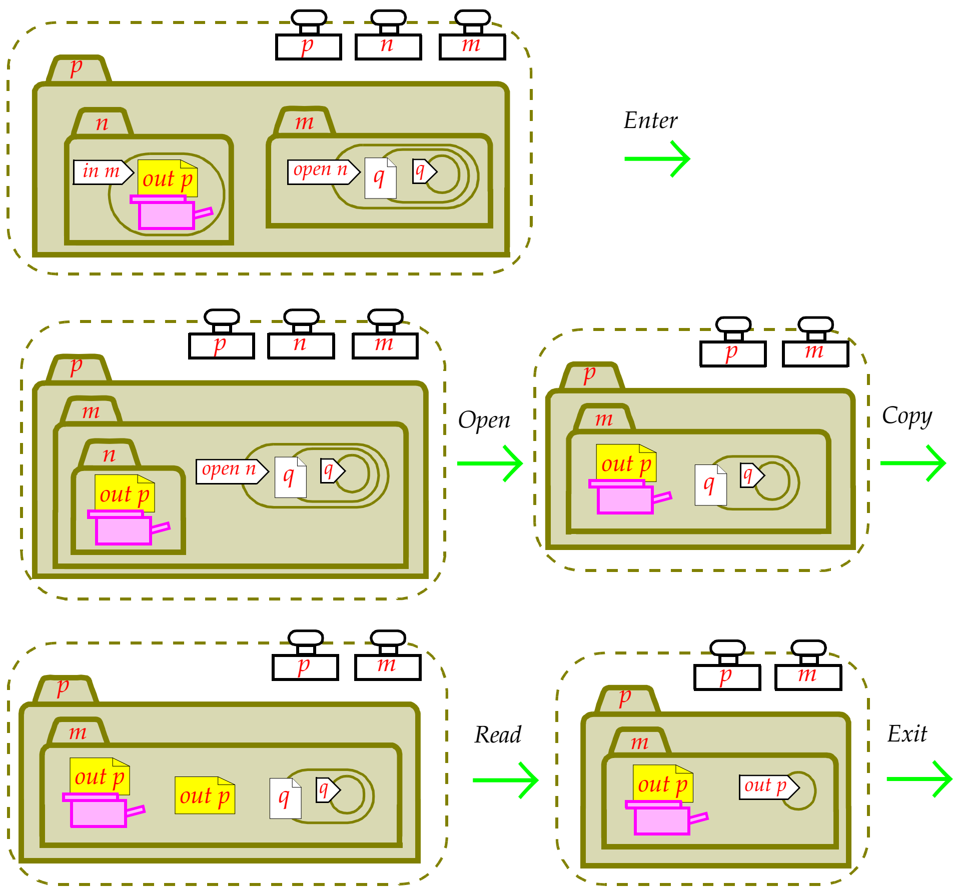
Naming capabilities are simply ambient names used as messages. A name n can be seen as a capability to construct (and rubber-stamp) a folder named n.

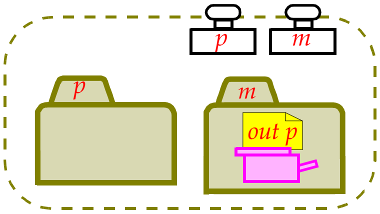
We have already seen the main navigation capabilities, implicitly. We have presented the operations *in n*, *out n* and *open n* as three distinct operations followed by a continuation P. In fact, they are special cases of a single operation $M.P$, where M is a navigation capability and P is the continuation after navigation. Given a name n, *in n* is the capability to enter an ambient named n, *out n* is the capability to exit an ambient named n, and *open n* is the capability to open an ambient named n. Navigation capabilities are extracted from naming capabilities, meaning that knowing n implies the ability to construct, for example, *in n*, but knowing *in n* does not imply knowing n.

Navigation capabilities can be composed into navigation paths. For example, *in n. out m. open p. out q* is a path capability that can be written in a single message, read into an input variable x, and executed in its entirety by $x.P$ (assuming, of course, that the path can be followed). It useful to have an empty navigation path, written *here*, such that *here.P* has no effect and continues with P, and *M.here* = *here.M* = M.

4.3.3 Example

This is an artificial example that uses each of the five folder reductions once. We use a single dashed line for multiple rubber stamps when the order of nesting of dashed lines does not matter. We remove rubber stamps when they are no longer needed.





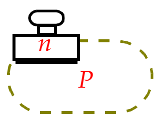
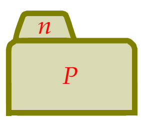
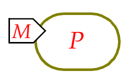
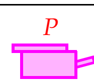
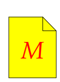
The reader is encouraged to follow the reductions, comparing them with their definitions.

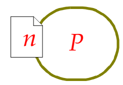
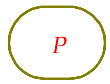
4.4 Textual Syntax

So far we have been relying on a visual syntax, but we are going to need some textual syntax for writing examples. The following table summarizes the folder calculus visual syntax, and introduces the corresponding textual syntax. There is a one-to-one correspondence between textual syntax and visual syntax; therefore, it is possible to freely mix them, if desired, nesting either one inside the other.

We use P,Q,R to range over ambient and process expressions, and M,N to range over message expressions. As shown in the table, the creation of a new name is written $(\nu n)P$ where the Greek letter ν (nu) binds the name n within the scope P. An ambient is written $n[P]$ where n is the ambient name, where the brackets denote the ambient boundary, and where P is the contents of the ambient.

Processes P,Q,R

Textual Syntax	***Visual Syntax***	***Comments***
$(\nu n)P$		New name n in a scope P.
$n[P]$		Folder (ambient) of name n and contents P.
$M.P$		Action M followed by P.
$P \mid Q$	P Q	Two processes in parallel. (Visually: contiguously placed in 2D.)
0	○	Inactive process (often omitted).
$!P$		Replication of P.
$\langle M \rangle$		Output M.

$(n).P$		Input n followed by P.
(P)		Grouping.

Messages M,N

n	A name
in n	An entry capability
out n	An exit capability
open n	An open capability
here	The empty path of capabilities
$M.N$	The concatenation of two paths

The textual syntax for the folder calculus is in fact the full syntax of the ambient calculus we were alluding to previously. Therefore, our folder calculus metaphor is quite exact: the syntax and semantics of our formal ambient calculus [10] has now been completely explained through the metaphor.

5 Ideas for Wide-Area Languages

The ambient calculus is a minimal formalism designed for theoretical study. Our final goal, though, is to program the Internet; for this we are certainly going to need something more elaborate and convenient than a formal calculus. Still, the basic constructs of the ambient calculus represent our understanding of the fundamental properties of mobile computation over wide-area networks. Therefore, we aim to find programming constructs that are semantically compatible with the principles of the ambient calculus.

5.1 Related Languages

Many software systems have explored and are exploring notions of mobility. Among these are:

- Obliq [9]. The Obliq language attacks the problems of remote execution and mobility for distributed computing. It was designed in the context of local area networks. Within its scope, Obliq works quite well, but is not really suitable for computation and mobility over the Web, just like most other distributed paradigms developed in pre-Web days.
- Telescript [30]. Our ambient model is partially inspired by Telescript, but is almost dual to it. In Telescript, agents move whereas places stay put. Ambients, instead, move whereas processes are confined to ambients. A Telescript agent, however, is itself a little ambient, since it contains a "suitcase" of data. Some nesting of places is allowed in Telescript.
- Java [18]. Java provides a working paradigm for mobile computation, as well as a huge amount of available and expected infrastructure on which to base more ambitious mobility efforts. The original Java mobility model, however was based on mobility of code, not mobility of data or active computations. Data mobility has now been achieved by the Java RMI extension, but computation mobility (e.g. for threads or live objects) is still problematic.
- Linda [11]. Linda is a "coordination language" where multiple processes interact in a common space (called a tuple space) by dropping and picking up tokens asynchronously. Distributed versions of

Linda exist that use multiple tuple spaces and allow remote operations over those. A dialect of Linda [12] allows nested tuple spaces, but not mobility of the tuple spaces.

5.2 Wide-Area Languages

We do not have a full-blown language or library to propose yet; what is more important here is the general flavor of such a language or library. In this section we discuss programming constructs that are directly inspired by the ambient calculus and that are not usually found in standard programming environments. In the remaining sections we show a detailed example involving some of these constructs, and we speculate on a specific wide-area application that could hopefully be better written using a wide-area language.

5.2.1 Ambients as a Programming Abstraction

Our basic abstraction is that of mobile computational ambients. The ambient calculus brings this abstraction to an extreme, by representing *everything* in terms of ambients at a very fine grain. In practice, to be useful as a programming abstraction, ambients would have to be medium or large-grained entities. Ambient contents should include standard programming subsystems such as modules, classes, objects, and threads.

In that sense, ambients could be used simply as wrappers of ordinary software subsystems, for the purpose of rendering those subsystems mobile. An ambient library could include mechanisms for creating empty ambients, adding threads, objects, and subambients to them, allowing threads to communicate with other ambients, and, of course, allowing ambients to move around.

We should notice that the ability to smoothly move a collection of running threads is almost unheard of in current software infrastructures. In this sense, ambients would be a novel and non-trivial addition to our collection of programming abstractions.

5.2.2 Names vs. Pointers

The only way to use an ambient is by its name. No matter how we organize our hierarchies of ambients, the only way to manipulate the resulting structure is by using ambient names. These names are detached from their corresponding ambients; one may possess a name without having immediate access to any ambient of that name.

Therefore, names are like pointers, providing access to structures, but are only "symbolic pointers": more like file names and URLs than memory addresses. Like file names, these pointers need not always denote the same structure: they denote any ambient of the corresponding name, and this may change over time. When these pointers denote no ambient, they are not "broken" (like dangling pointers, or illegal file names) but rather "blocked" until a suitable ambient becomes available. Of course, one might feel nervous about the possibility of blocking whenever one tries to use a name. On the other hand, a blocked computation can be unblocked by providing an appropriate ambient, thus modeling both dynamic linking and the installation of plug-ins.

In an ambient-based language, every pointer to a data structure or other resource outside of a given ambient should behave like a name. This is necessary to allow ambients to move around freely without being restrained by immobile ties.

5.2.3 Locations

Ambients can be used to model both physical and virtual locations. Some physical locations are mobile (such as airplanes) while others are immobile (such as buildings). Similarly, some virtual locations are mobile (such as agents) while others are immobile (such as mainframe computers). These mobility distinctions are not reflected in the semantics of ambients, but can be added as a refinement of the basic model, or embedded in type systems that restrict the mobility of certain ambients.

5.2.4 Migration and Transportation

Each ambient is completely self-contained, and can be moved at any time with all its running computations.

If an ambient encloses a whole application, then the whole running application can be moved without need to restart it or reinitialize it. In practice, an application will have ties to the local window system, the local file system, etc. These ties, however, should only be via ambient names. When moving the applications, the old window system ambient, say, becomes unavailable, and eventually the new window system ambient becomes available. Therefore, the whole application can smoothly move and reconnect its bindings to the new local environment. Some care will still be needed to restart in a good state (say, to refresh the application window), but this is a minor adjustment compared to what one would have to do if hard connections existed between the application and the environment [4].

5.2.5 Communication

The basic communication primitives of the ambient calculus are based on the asynchronous model and do not support global consensus or failure detection. These properties should be preserved by any higher-level communication primitives that may be added to the basic model, so that the intended semantics of communication over a wide-area networks is preserved.

The ambient calculus directly supports only local communication within an ambient. Remote communication (for example, RPC) can be interpreted as mobile output packets that transport and deposit messages to remote locations, wait for local input, and then come back. The originator of an RPC call may block for the answer to come back before proceeding, in the style of a synchronous call. In this interpretation, the outgoing and incoming packets may get stuck for an arbitrary amount of time, or get lost. There may be no indication that the communication has failed, and therefore the invoking process may block forever without receiving a communication exception. This is as it should be, since arbitrary delays are indistinguishable from failures. (Note, though, that a time-out mechanism is easily implemented, by placing a remote invocation in parallel with another activity that waits a certain time and, if the invocation has not completed, cause something else to happen.)

Other examples of derived communication mechanisms include parent-child communication, and communication between siblings (perhaps aided by the common parent). All these appear quite useful, and will likely need to be included in any convenient language.

5.2.6 Data Structures

Basic data structures, such as booleans and integers, can be encoded in terms of ambients, but the encodings are not practical. Therefore, basic types should be taken as primitive, as usual.

Ambients can directly express hierarchies, so it should not be surprising that they can easily represent structured data types. For example, a record structure of the form $\{l_1{=}e_1, ..., l_n{=}e_n\}$ can be represented as:

$$r[l_1[!(\langle e_1\rangle \mid open\ accessor)] \mid ... \mid l_n[!(\langle e_n\rangle \mid open\ accessor)]]$$

where the name r is the address of the record, and the record fields $l_i{=}e_i$ are represented by subambients of the record ambient. The values e_i are placed into the subambients as outputs, waiting for an accessor ambient to go inside and read them.

The ambient semantics, though, is a bit richer than the minimum required for data structures. For example, we would not normally want to allow the fields of a record to take off and leave. Some possibilities supported by the ambient semantics are intriguing but probably not worth the trouble, such as the ability to concatenate records by opening them inside a new record.

Therefore, it would be prudent to build-in ordinary data structures, without relying too much on the possibilities provided by the ambient semantics. Still, for ambients to work at all we need these data structures to be mobile. Therefore, the "address" of a data structure should be a proper ambient name, in the sense that it should provide blocking behavior on access if needed.

5.2.7 Recursion

Many behaviors are naturally defined recursively. It is convenient to use a recursion construct of the form, for example, *rec* $X.\ P$, representing a recursively defined process P where any occurrence of X again denotes

P. More generally, one can introduce mutually recursive definitions, as done below when discussing resources.

The ambient calculus embeds what is essentially an iteration construct, $!P$. In terms of theoretical power, this is sufficient to encode recursion. As a programming construct, though it is inconvenient, and is also hard to implement because of its flavor of unbounded generation. So, for programming practice $!P$ should be removed in favor of recursion facilities.

5.2.8 Synchronization

Synchronization primitives are needed to coordinate the activities of multiple processes within an ambient. In the ambient calculus, it is easy to represent basic synchronization constructs, such as mutexes:

$\mathit{release}\ n; P \triangleq n[] \mid P$	release a mutex called n, and do P
$\mathit{acquire}\ n; P \triangleq \mathit{open}\ n.\ P$	acquire a mutex called n, then do P

where the *open* instruction blocks until a mutex is released, and consumes the mutex.

A useful technique is to synchronize on the change of name of an ambient. That is, a process may wait for an ambient to change its name to a given name, and another process may perform the change to trigger the first process. Name change can be represented in the ambient calculus, but not in a way that is atomic with respect to arbitrary actions (particularly, movement) performed by other processes. A proper solution would require all the processes in an ambient to synchronize on a mutex before any movement or name change. This is a bit laborious; therefore it is convenient to introduce name change as a primitive with the following atomic reduction:

$$n[\mathit{be}\ m.P \mid Q] \rightarrow m[P \mid Q]$$

5.2.9 Static and Dynamic Binding

Wide-area systems are outside the control of any single administrator, and therefore their maintenance cannot rely entirely on static configurations. In programming language terms, they cannot rely completely on static binding of variables. A programer who needs to change a static binding can usually change it, recompile the program, and restart it. But one cannot easily restart the Internet, or any large system deployed across it. Therefore some form of dynamic binding is necessary.

The names of the ambient calculus represent an unusual combination of static and dynamic binding. Formally, the names obey the classical rules of static scoping, including consistent renaming, capture-avoidance, and block nesting. However, the navigation primitives behave by dynamically binding a name to any ambient that has the right name. The connection between a name and an ambient of that name is fully dynamic, much like in dynamic linking.

The basic ambient calculus does not have any definitional facilities, but these are clearly needed when building any large-scale systems, or even medium scale examples. We are going to need definitional facilities for both static and dynamic binding, corresponding to the underlying semantic features.

Statically-bound definitions can be represented by fairly normal *let* constructions. A process-producing definition can be written:

$\mathit{let}\ f(x_1...x_n) = P; P'$ (let $f(x_1...x_n)$ be P within the scope P')

with a standard expansion semantics within the scope P'. For call-by-value parameter passing, assuming that $V_1...V_n$ are fully-evaluated messages, we have:

$$f(V_1...V_n) \triangleq P\{x_1\leftarrow V_1...x_n\leftarrow V_n\}$$

Dynamically bound definitions are much less routine, and more closely tied to the ambient semantics; we discuss them next.

5.2.10 Resources

In the mobile agent paradigm, sites differ from each other by the collection of resources that they provide to visiting agents. Agents bind dynamically to these resources as they enter the ambient. Similarly, it is natural to think of each ambient as providing a collection of local resources, which could be seen as forming the interface, namespace, environment, method suite, etc., of the ambient.

To this end, we introduce a mechanism of local resource definitions that is compatible with the general ambient semantics, and can be easily explained in terms of the primitives of section 4. These resource definitions may occur only at the top level of an ambient, and have the following form:

$$def\ f(x_1 ... x_n) = P; \qquad \text{(bind the resource } f(x_1 ... x_n) \text{ to } P \text{ within the ambient)}$$

Note that the name *f* is not bound by this definition; it must be bound by a ν binder at some sufficiently global level. This way, different resources named *f* can be provided in different ambients, and each use of the resource *f* will be relative to a given ambient. A typical such resource could be the local window-system library.

A resource *f* can be invoked as follows, within an ambient where it is defined:

$$f(V_1 ... V_n)/here \;\triangleq\; P\{x_1{\leftarrow}V_1 ... x_n{\leftarrow}V_n\}$$

where the suffix "*/here*" indicates that a definition of *f* has to be found in the ambient where *f* is invoked. In general, a path may be used in place of *here*, in which case the definition of the resource is retrieved from the ambient obtained by following the path, and the invocation is executed there. In other words, the effect of $f(V_1 ... V_n)/p$ is to transport the invocation $f(V_1 ... V_n)$ along the path p, and then to invoke it within the target ambient. This can be expressed by two rules that each transport an invocation one step further:

$$f(V_1 ... V_n)/in\ n.p \mid n[Q] \;\triangleq\; n[Q \mid f(V_1 ... V_n)/p]$$
$$n[f(V_1 ... V_n)/out\ n.p \mid Q] \;\triangleq\; f(V_1 ... V_n)/p \mid n[Q]$$

For example, we could have an expression such as the following:

$$\begin{array}{l} n[def\ x() = \langle 1 \rangle; \\ \quad def\ f(y) = x()/here \mid (x').\ \langle x'+y \rangle] \\ \mid f(3)/in\ n \end{array}$$

producing, after the invocation $f(3)/in\ n$:

$$\begin{array}{l} n[def\ x() = \langle 1 \rangle; \\ \quad def\ f(y) = x()/here \mid (x').\ \langle x'+y \rangle; \\ \quad \langle 4 \rangle] \end{array}$$

The definition and invocation of resources can be encoded in the pure ambient calculus; this way, resources automatically acquire an ambient-like flavor. In particular, if an ambient has no definition for an invocation, the invocation blocks until the definition becomes available in the designated place. If an ambient has multiple definitions for an invocation, any one of them may be used. If an ambient containing definitions is opened, its definitions become definitions of its parent.

5.2.11 Modularization

An ambient *P* that includes a collection of local definitions can be seen as a module, or a class. More precisely, !*P* can be seen as a module or a class (since *P* there is inactive), while any active *P* generated from it can be seen as a module instance or an object.

The action of performing an *open* on such an ambient can be seen as importing from a module or inheriting from a class, since ambient definitions are transplanted from one ambient to another. Moreover, one can regard the notation $f(M)/p$ either in module-oriented style as $p.f(M)$ (the invocation of $f(M)$ from the module at p), or in object-oriented style as *delegation* [29] of $f(M)$ to the object found at p.

When seen as modules or components, ambients have several interesting and unusual properties.

First, ambients are *first class modules*, in the sense that one can choose at run time which particular instance of a module to use.

Second, ambients support *dynamic linking*: missing subsystems can be added to a running system simply by placing them in the right spot.

Third, ambients support *dynamic reconfiguration*. In most module and class systems, the identity of individual modules is lost immediately after static or dynamic linking. Ambients, though, maintain their identity at run time. As a consequence, a system composed of ambients can be reconfigured by dynamically replacing an ambient with another one. The blocking semantics of ambient interactions allows the system to smoothly suspend during a configuration transition. Moreover, the hierarchical nature of ambients allows the modular reconfiguration of entire subsystem, not just of individual modules. Dynamic reconfiguration is particularly valuable in long-running and widely-deployed systems that cannot be easily stopped (for example, in telephone switches); it is certainly no accident that thinking about the Internet led us to this property.

Therefore, ambients can be seen as proper *software components*, according to a paradigm often advocated in software engineering, where the components are not only replaceable but also mobile.

5.2.12 Security

Ambient security is based on capabilities and on the notion that security checks are performed at ambient boundaries, after which processes are free to execute until they need to cross another boundary. This is a capability-based model of security, as opposed to a cryptography-based model, or an access-control based model.

These three models are all interdefinable. In our case, access control is obtained by using ambients to implement RPC-like invocations that have to cross boundaries and authenticate every time. The cryptographic model is obtained by interpreting encryption keys as ambient names, which are by assumption unforgeable. Then, encryption is given by wrapping some content in an ambient of a given name, and decryption is obtained by either entering or opening such an ambient given an appropriate capability (the decryption key).

5.2.13 Summary

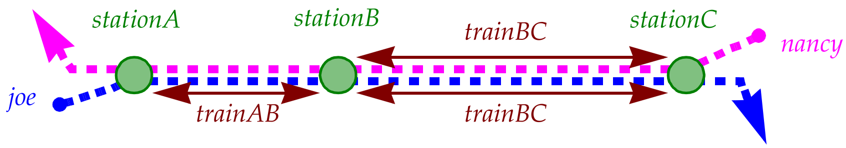
We believe we have sufficiently illustrated how the ambient semantics naturally induces unusual programming constructs that are well-suited for wide-area computation. The combination of mobility, security, communication, and dynamic binding issues has not been widely explored yet at the language-design level, and certainly not within a unifying semantic paradigm. We hope our unifying foundation will facilitate the design of such new languages.

5.3 Example: Public Transportation

We now show an example of a program written in ambient notation. Some additional constructs used here have been introduced in the previous section.

This example emphasizes the mobility aspects of ambients, and the fact that an ambient may be transported from one place to another without having to know the exact route to be followed. A passenger on a train, for example, only needs to know the destination station, and need not be concerned with the intermediate stations a train may or may not stop at.

In this example, there are three train stations, represented by ambients: *stationA*, *stationB* and *stationC* (of course, these particular ambients will never move). There are three trains, also represented by ambients: a train from *stationA* to *stationB* originating at *stationA*, and two trains between *stationB* and *stationC*, one originating at each end. There are two passengers, again represented by ambients; *joe* and *nancy*. *Joe* wants to go from *stationA* to *stationC* by changing trains at *stationB*; *nancy* wants to go the other way.



We begin by defining a parametric process that can be instantiated to trains going between different stations at different speeds. The parameters are: *stationX*: the origin station; *stationY*: the destination station; *XYatX*, the tag that the train between *X* and *Y* displays when stationed at *X*; *XYatY*, the tag that the train between *X* and *Y* displays when stationed at *Y*; *tripTime*, the time a train takes to travel between origin and destination.

The definition of a train begins with the creation of a new name, *moving*, that is intermittently used as the name of the train. While the train is *moving*, passengers should not be allowed to (dis)embark; this is achieved by keeping *moving* a secret name. The train begins as an ambient with name *moving*, and contains a single recursive thread that shuttles the train back and forth between two stations. Initially, the train declares itself to be a train between *X* and *Y* stationed at *X*, and waits some time for passengers to enter and exit. Then it becomes *moving*, so passengers can no longer (dis)embark. It exits the origin station, travels for the *tripTime*, enters the destination station, and declares itself to be the train between *X* and *Y* at *Y*. Again, passengers can (dis)embark during the wait time at the station. Then the train becomes *moving* again, goes back the other way, and then repeats the whole process.

let train(*stationX stationY XYatX XYatY tripTime*) =
    (ν *moving*)   // assumes the train originates inside *stationX*
        moving[*rec T*.
            be XYatX. *wait* 2.0.
            be moving. *out stationX*. *wait tripTime*. *in stationY*.
            be XYatY. *wait* 2.0.
            be moving. *out stationY*. *wait tripTime*. *in stationX*.
            T];

Next we have the configuration of stations and trains. We create fresh names for the three stations and for the train tags. Then we construct three ambients for the three stations, each containing an appropriately instantiated train.

(ν *stationA stationB stationC ABatA ABatB BCatB BCatC*)
    stationA[*train*(*stationA stationB ABatA ABatB* 10.0)] |
    stationB[*train*(*stationB stationC BCatB BCatC* 20.0)] |
    stationC[*train*(*stationC stationB BCatC BCatB* 30.0)] |

Finally, we have the code for the passengers. *Joe*'s itinerary is to enter *stationA*, wait to enter the train from *A* to *B* when it is stationed at *A*, exit at *B*, wait for the train from *B* to *C* when it is stationed at *B*, exit at *C* and finally exit *stationC*. During the time that *joe* is waiting to exit a train, he is blocked waiting for the train to acquire the appropriate tag. The train could change tags at intermediate stations, but this would not affect *joe*, who is waiting to exit at a particular station. When that station is reached, and the train assumes the right tag, *joe* will attempt to exit. However, there is no guarantee that he will succeed. For example, *joe* may have fallen asleep, or there may be such a rush that *joe* does not manage to exit the train in time. In that case, *joe* keeps shuttling back and forth between two stations until he is able to exit at the right station.

(ν *joe*)
joe[
in stationA.
in ABatA. *out ABatB*.
in BCatB. *out BCatC*.
out stationC] |

The code for *nancy* is similar, except that she goes in the other direction. Given the timing of the trains, it is very likely that *nancy* will meet *joe* on the platform at *stationB*.

(ν *nancy*)
nancy[
in stationC.
in BCatC. *out BCatB*.
in ABatB. *out ABatA*.
out stationA]

In all this, *joe* and *nancy* are active ambients that are being transported by other ambients. Sometimes they move of their own initiative, while at other times they move because their context moves. Note that there are two trains between *stationB* and *stationC*, which assume the same names when stopped at a station. *Joe* and *nancy* do not care which of these two train they travel on; all they need to know is the correct train tag for their itinerary, not the "serial number" of the train that carries them. Therefore, having multiple ambients with the same name simplifies matters.

When all these definitions are put together and activated, we obtain a real-time simulation of the system of stations, trains, and passengers. A partial trace looks like this:

nancy: moved in *stationC*
nancy: moved in *BCatC*
joe: moved in *stationA*
joe: moved in *ABatA*
joe: moved out *ABatB*
nancy: moved out *BCatB*
joe: moved in *BCatB*
nancy: moved in *ABatB*
nancy: moved out *ABatA*
nancy: moved out *stationA*
joe: moved out *BCatC*
joe: moved out *stationC*

5.4 Challenge: A Conference Reviewing System

We conclude with the outline of an ambitious wide-area application. The application described here does not fit well with simple-minded Web-based technology because of the complex flow of active code and stateful information between different sites, and because of an essential requirement for disconnected operation. The application fits well within the agent paradigm, but also involves the traversal of multiple administrative domains, and has security and confidentiality requirements.

This is meant both as an example of an application that could be programmed in a wide-area language, and as a challenge for any such language to demonstrate its usability. We hope that a language based on ambients or similar notions would cope well with this kind of situation.

5.4.1 Description of the problem

The problem consists in managing a virtual program committee meeting for a conference. The basic architecture was suggested to me by comments by Richard Connors, as well as by my own experience with organizing program committee meetings and with using Web-based reviewing software developed for ECOOP and other conferences.

In the following scenario, the first occurrence of each of the principals involved is shown in boldface.

Announcement

A **conference** is announced, and an electronic **submission form**, signed by the **conference chair**, is publicized.

Submission

Each **author** fetches the submission form, checks the signature of the conference chair, and activates the form. Once activated, the form actively guides most of the reviewing process. Each author fills an instance of the form and attaches a **paper**. The form checks that none of the required fields are left blank, electronically signs the paper with a signature key provided by the author, encrypts the attached paper, and finds its way to the **program chair**. The program chair collects the submissions forms, and gives them a decryption key so that they can decrypt the attached papers and verify the signatures of the authors. (All following communications are signed and encrypted; we omit most of these details from now on.)

Assignment

The program chair then assigns the submissions to the **committee members**, by instructing each submission form to generate a **review forms** for each assigned member. The review forms incorporate the paper (this time signed by the program chair) and find their way to the appropriate committee members.

Review

Each committee member is a **reviewer**, and may decide to review the paper directly, or to send it to another reviewer. The review form keeps tracks of the chain of reviewers so that it can find its way back when either completed or refused, and so that each reviewer can check the work of the subreviewers. Eventually a review is filled. The form performs various consistency checks, such as verifying that the assigned scores are in range and that no required fields are left blank. Then it finds its way back to the program chair.

Report generation

Once the review forms reach the program chair, they become **report forms**. The various report forms for each paper merge with each other incrementally to form a single report form that accumulates the scores and the reviews. The program chair monitors the report form for each paper. If the reviews are in agreement, the program chair declares the form an **accepted paper report form**, or a **rejected paper review form**.

Conflict resolution

If the reports are in disagreement, the program chair declares the form an **unresolved review form**. An unresolved review form circulates between the reviewers and the program chair, accumulating further comments, until the program chair declares the paper accepted or rejected.

Notification

The report form for an accepted or rejected paper finds its way back to the author (minus the confidential comments), with appropriate congratulations or regrets.

Final versions

Once it reaches the author, an accepted paper report form spawns a **final submission form**. In due time, the author attaches to it the final version of the paper and signs the copyright release notice. The completed final submissions form finds its way back to the program chair.

Proceedings

The final submission forms, upon reaching the program chair, merge themselves into the **proceedings**. The program chair checks that all the final versions have arrived, sorts them into a conference schedule, attaches a preface, and lets the proceedings find their way to the conference chair.

Publication

The conference chair files the copyright release forms, signs the proceedings, and posts them to public sites.

5.4.2 Comments

A few critical features characterize this application as particularly well-suited for experimenting with wide-area languages.

First, it is a requirement of this application that most interactions happen in absence of connectivity. Virtual committee meetings occur over the span of one month of reviewing and one or two weeks of discussion. It is highly unlikely that all the committee members will be continuously near their main workstation, or any workstation, during such a span of time. Yet, progress cannot be interrupted by the temporary absence of any one member. Furthermore, progress cannot be interrupted by the absence of connectivity for any one member: paper reviews are commonly done on airplanes, in doctors waiting rooms, in lines at the Post Office, in cafe's, etc. While a laptop or personal organizer can be easily carried in those environments, continuous connectivity is far from easy to achieve. This is to be contrasted with current web-based review systems, which require reviewers to sit at a connected workstation while filling the review forms.

Second, form-filling requires semantic checking, which is best done while the form is being filled. Therefore, active forms are required even during off-line operation. This is to be contrasted with the filling of on-line Web-based review forms which require, in practice, preparing reviews off-line on paper or in ASCII, and later typing them or pasting them laboriously into on-line forms in order to obtain the semantic checking. Alternatively, if the review is simply emailed in ASCII, then the program chair has the considerable burden of performing the parsing and semantic checking.

Third, unattended operations is highly desirable also for the program chair. The program chair may go in vacation after the assignment phase and come back to find all the report forms already merged, thanks to the use of active forms.

Fourth, the system must handle multiple administrative domains. Committee members are intentionally selected to belong to widely diverse and dispersed institutions, many of which are protected by firewalls. In this respect, this situation is different from classical office workflow on a local area network, although it shares many fundamental features with it.

Fifth, all the forms are active. This relieves various principals from the tedious and error-prone task of collecting, checking, and collating pieces of information, and distributing them to the correct sets of other principals.

In summary, in this example, interactions between various parts of the system happen over a wide-area network. The people involved may be physically moving during or between interaction. As they move, they may transport without warning active parts of the system. At other times, active parts of the system move by their own initiative and must find a route to the appropriate principals wherever they are.

6 Conclusions

The global computational infrastructure has evolved in fundamental ways beyond standard notions of sequential, concurrent, and distributed computational models. *Mobile ambients* capture the structure and properties of wide-area networks, of mobile computing, and of mobile computation. The ambient calculus [10] formalizes these notions simply and powerfully. It supports reasoning about mobility and security, and has an intuitive presentation in terms of the folder calculus. On this basis, we can envision new programming methodologies, libraries and languages for global computation.

7 Acknowledgments

The ideas presented in this paper originated in the heated atmosphere of Silicon Valley during the Web explosion, and were annealed by to the cool and reflective environment of Cambridge UK. I am deeply indebted to people in both locations, particularly to Andrew Gordon, who is also a coauthor of related papers.

References

[1] Abadi, M. and A.D. Gordon, **A calculus for cryptographic protocols: the spi calculus**. *Proc. of the Fourth ACM Conference on Computer and Communications Security*, 36-47, 1997.

[2] Agha, G. A., **Actors: A Model of Concurrent Computing in Distributed Systems**, MIT Press, 1986.

[3] Amadio, R.M., **An asynchronous model of locality, failure, and process mobility**. *Proc. COORDINATION 97*, Lecture Notes in Computer Science 1282, Springer Verlag. 1997.

[4] Bharat, K. and L. Cardelli: **Migratory applications**, *Proc. of the ACM Symposium on User Interface Software and Technology '95*. 133-142. 1995.

[5] Berry, G. and G. Boudol, **The chemical abstract machine**. *Theoretical Computer Science* **96**(1), 217-248, 1992.

[6] Boudol, G., **Asynchrony and the p-calculus.** *Technical Report 1702, INRIA, Sophia-Antipolis,* 1992.

[7] Bracha, G. and S. Toueg, **Asynchronous consensus and broadcast protocols**. *J.ACM* **32**(4), 824-840. 1985.

[8] Brauer, W., ed., **Net Theory and Applications**, *Proc. of the Advanced Course on General Net Theory of Processes and Systems, Hamburg, 1979.* Lecture Notes in Computer Science 84. Springer-Verlag. 1980.

[9] Cardelli, L., **A language with distributed scope**. *Computing Systems,* **8**(1), 27-59. MIT Press. 1995.

[10] Cardelli, L., A.D. Gordon, **Mobile Ambients**, in *Foundations of Software Science and Computational Structures*, Maurice Nivat (Ed.), Lecture Notes in Computer Science 1378, Springer, 140-155. 1998.

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

[12] Carriero, N., D. Gelernter, and L. Zuck, **Bauhaus Linda**, in *Object-Based Models and Languages for Concurrent Systems*, P. Ciancarini, O. Nierstrasz and A. Yonezawa (Ed.), Lecture Notes in Computer Science 924, Springer Verlag, 66-76. 1995.

[13] Chandra, T.D., S.Toueg, **Unreliable failure detectors for asynchronous systems.** *ACM Symposium on Principles of Distributed Computing*, 325-340. 1991.

[14] De Nicola, R., G.-L. Ferrari and R. Pugliese, **Locality based Linda: programming with explicit localities**. *Proc. TAPSOFT'97*. Lecture Notes in Computer Science 1214, 712-726, Springer Verlag. 1997.

[15] Ficher, M.J., N.A. Lynch, and M.S. Paterson, **Impossibility of distributed Consensus with one faulty process.** *J.ACM* **32**(2), 374-382. 1985.

[16] Fournet, C. and G. Gonthier, **The reflexive CHAM and the join-calculus**. *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, 372-385. 1996.

[17] Fournet, C., G. Gonthier, J.-J. Lévy, L. Maranget, D. Rémy, **A calculus of mobile agents**. *Proc. 7th International Conference on Concurrency Theory (CONCUR'96)*, 406-421. 1996.

[18] Gosling, J., B. Joy and G. Steele, **The Java language specification**. Addison-Wesley. 1996.

[19] Hoare, C.A.R., **Communicating sequential processes**. *Communications of the ACM* **21**(8), 666-678. 1978.

[20] Honda., K. and M. Tokoro, **An object calculus for asynchronous communication.** *Proc. ECOOP'91,* Lecture Notes in Computer Science 521, 133-147, Springer Verlag, 1991.

[21] INMOS Ltd., **occam programming manual**. Prentice Hall. 1984.

[22] Milner, R., **A calculus of communicating systems**. Lecture Notes in Computer Science 92. Springer Verlag. 1980.

[23] Milner, R., **Functions as processes**. *Mathematical Structures in Computer Science* **2**, 119-141. 1992.

[24] Milner, R., J. Parrow and D. Walker, **A calculus of mobile processes, Parts 1-2**. *Information and Computation*, **100**(1), 1-77. 1992

[25] Morris, J.H., **Lambda-calculus models of programming languages.** Ph.D. Thesis, MIT, Dec 1968.

[26] Palamidessi, C., **Comparing the expressive power of the Synchronous and the Asynchronous pi-calculus**. *Proc. 24th ACM Symposium on Principles of Programming Languages*, 256-265. 1997.

[27] Sander, A. and C. F. Tschudin, **Towards Mobile Cryptography**, *ICSI technical report 97-049*, November 1997. *Proc. IEEE Symposium on Security and Privacy*, Spring 1998 (to appear).

[28] Sangiorgi, D. **From π-calculus to Higher-order π-calculus - and back**, *Proc. TAPSOFT '93.*, Lecture Notes in Computer Science 668, Springer Verlag. 1992.

[29] Stein, L. A., Lieberman, H., and Ungar, D. 1988. **A shared view of sharing: The treaty of Orlando**. In *Object-oriented concepts, applications, and databases*, W. Kim and F. Lochowsky, eds., 31-48. Addison-Wesley.

[30] White, J.E., **Mobile agents**. In *Software Agents*, J. Bradshaw, ed. AAAI Press / The MIT Press. 1996.