A Language-Based Approach to Programmable Networks

Ian Wakeman Alan Jeffrey
Tim Owen
School of Cognitive and Computer Science, University of Sussex, BRIGHTON, UK

November 29, 1999

Abstract

It appears that some degree of programmability is inevitable within the network, whether it be through active networks, active services, or programmable middleware. We argue that programming network elements with languages designed for use within a single machine is inappropriate, since the only defense for the shared resource of the network is through the use of sandboxes, which are prone to performance problems and are difficult to implement correctly. Instead, we believe that new languages should be designed for programmable networks, using type systems that ensure safe programs, and encourages correct programs.

We have designed and provided the full semantics for such a language. Building upon this, we have implemented a compiler, runtime environment and a simulation environment for our language. In this paper we describe the major features of the language that protect the network: abstracted locations; located objects; volatile routing; thread and class loading; and enforced resource counting. We show how these features are used in a number of small case studies, and in implementing optimised communication libraries. The ease with which these demonstrations have been built and debugged shows the potential for enforcing network programming models with well-typed languages.

1 Introduction

A key strength of the Internet has been to reduce the intelligence within the network to that required to maintain connectivity through the routing tables in the routers. This has enabled the network to grow quickly and to scale with a reasonable amount of comfort. However, new applications are forcing the network architects to reassess the functionality within the network. Audio and video have forced the introduction of some per-flow (or group of flows) state within the router [1, 2, 3, 4]. Firewalls have been extended to allow programming of the filters, requiring state machines to operate upon the received packets. Nearly all forms of group communication, whether it be based upon multicast and explicit message delivery or on more implicit mechanisms such as sharing information at a Web proxy, have used protocol specific code running on nodes within the network to connect the group members together in various topologies.

The research community has risen to the challenge of network programmability by defining a number of extensions to the network architecture. Some have chosen to develop middleware which uses tried and trusted technology such as RPC and CORBA to offer services on routers within the network [5]. However, we believe that RPC-based mechanisms suffer from problems of latency in the communication between application and service element, and will not be effective for many applications.

An alternative approach has been to assume that applications can run their own code on some public processing elements within the network. This increases the flexibility of the architecture, since the code can then react to changes in the environment on the spot, rather than suffering an RPC round trip time before reacting. These processing nodes are placed at strategic places within the network such as at administrative boundaries, or at locations where there is a large mismatch between bandwidth, or where the certain location specific services can be used. Work within this area ranges from the Active Services of McCanne et al [6] through aglets from IBM [7] to proxylets [8].

Others believe that the entire architecture should be rethought as a computational environment, in which everything can be programmed, and the entire network becomes *active* [9]. The research in this area is aimed at discovering how viable it is to open up such elements of the communications architecture as the routing table.

A fundamental question raised by both the active service and the active network approaches is how to ensure that the shared resource of the network remains safe and is protected from misbehaving programs. Programs can abuse the network by generating packet explosions and can abuse the shared processor by using all the memory and the processor cycles. Worse, they may subvert the working of correct programs so that they too break. If network programmability is going to be available to the application designers, we need to ensure that they do not break things by accident, let alone by intention.

Traditional systems approaches to protection are based upon defining what a program should be able to do, then using runtime checks to ensure that the program doesn't exceed these bounds. This leads to the sandbox model of protection, as used in Java and enhanced in [10] to provide protection for Active Networks.

However, there are major problems with this approach. First, each runtime check reduces the performance of the system, increasing the overhead of each use of system resources. Second, it is very difficult to ensure that the protection mechanisms are correct, and cannot be subverted in any way.

An alternative approach is to use compile time checks upon what the program is doing. This uses the type system to represent predicates about program functionality and if a program is well-typed, then it proves the program to obey the policies implemented in the type system. This approach has been used to allow users to run programs within the kernel as in Spin [11], and in protecting access to router functionality in the Switchware project [12]. By implementing protection policies in the type system, programs are checked for correctness at compile time, and so many runtime checks can be removed. It is still a difficult task to prove that the type system correctly implements the protection policies, but at least this problem is now punted to the language designers, who have the mathematical tools to prove the semantics of the language.

In this paper we describe our design of the SafetyNet programming language and environment. SafetyNet is a well-typed language for use in Active Networks and Active Service architectures. The type system is designed to ensure correct programs cannot destroy network integrity. In doing so, we have converted the problem of protocol design into the problem of designing distributed applications. The work has grown out of our previous work in designing generalisable proxies for mobile clients [13, 14], in which we adapt the implementation of the performance by substituting different behaviours for different resource constraints, using the techniques of Open Implementation [15]. The language most similar to our approach is the PLAN language within the Switchware project [16]. However, the PLAN language is functional in approach, whereas we use a class-based object-oriented approach. We believe that more programmers will be comfortable with an OO language.

The presentation of this paper focuses upon the system aspects of the language, explaining how the architectural requirements have influenced the design of the language. The semantics of the language can be viewed online at `http://klee.cs.depaul.edu/an/spec/`, whilst a paper descibing the core features of the semantics has been submitted for publication [17].

The rest of the paper is as follows: In section 2 we describe the language and how it protects the network. We then describe how the language can be used to code a simple remote procedure call. We conclude with descriptions of work currently in progress and future work.

2 SafetyNet Design

The design goals of the SafetyNet language are:

- To provide a network programming language based on Internet 'best effort' communication.
- To provide scaleable high-level communication based on 'remote spawn' from which other communication can be built.
- To make use of types as safety properties, to ensure that the safety and security policies of the network are maintained.
- To rapidly prototype tools such as compilers and simulators in order to drive the development of the language by examples.

In this section, we discuss how these goals have led to the development of the prototype SafetyNet language.

2.1 Best-effort distributed programming

In the Internet, an application transmits a packet, which is sent to the next router on the way to the destination. At this router, the arrival of the packet causes code to run, which calls other code dependent upon the fields in the header of the packet. This code may access and modify local state stored in the router and then copy or create one or more packets to be sent out from the router. These packets are then routed on output links depending upon the destination for each packet, and so on until the packets reach their destination, or are destroyed within the network for whatever reason.

In our programming model, we have attempted to replicate this basic structure of packet transmission. In the Internet, the arrival of a packet initiates some thread of control which uses the data within the packet to decide upon the disposition of the packet. In our model, a packet becomes a thread of control, carrying the code to be run and the names or values of any data referenced within that code. When a thread arrives at a Safetynet-aware router or end system, the thread code is instantiated within the runtime and runs within a defined scheduling class. The thread of control may call other code to be run on its behalf. The other code is encapsulated within classes, which are either present in the router, or are dynamically loaded from elsewhere. Threads can spawn other threads, either locally or on the next hop to some destination.

In this model, the atomic unit of communication is to remotely spawn a thread at the next hop to a destination `dest`:

```
spawn towards dest {
  ...code...
}
```

Note that there is *no* return value from this spawning. There is no direct acknowledgement signal from the next hop indicating that the spawn was successful or not. In this way, we model the best effort delivery of IP.

Also note that there are no guarantees that two such spawns will spawn a thread at the same location. The routing architecture of the Internet offers no guarantee that routes will remain constant. We reflect this dynamic routing in forcing the programmer to request a route to a destination, rather than allowing them to define which hop to route to next. We believe that IP routing is robust and well-tested, and that SafetyNet code should not have the ability to subvert the underlying routing mechanisms.

The primitive we have supplied to the programmer is remote spawning on the next hop to a location. This is a very low-level primitive, but from it we can build unicast communication, multicast communication, RPC, and many other communication styles.

There are two phases to implementing remote spawning: at *compile time*, we wrap the code up into a method call; and at *run time*, we serialize the data required for the method call, and then call the method, which may in turn require some class loading. For example:

```
let x : int = 99;
spawn towards dest {
  ...do something with x...
}
```

can be *closure converted* at compile-time to:

```
class Tmp {
  static method run (x : int) {
    ...do something with x...
  }
}
let x : int = 99;
spawn at route dest { Tmp.run (x); }
```

Remote spawning is thus just syntax sugar for an unreliable remote procedure call, but without any return values. At run-time, we execute this by serializing the data `x` and then calling `Tmp.run` on the serialized data. If this is the first time the code is executed, this will cause the `Tmp` class to be loaded, but the class can be cached for subsequent efficient execution, as occurs in the ANTS system.

2.2 Type Parameterisation

In constructing protocols, the actual data carried by the protocol packets is often opaque to the protocol. However, when the data pops out from the protocol, applications need to understand the exact structure of the data. Rather than use casting to ensure that the language respects the opacity of the data, we have used generic class parameters, and F-bounded polymorphism, similar to the implementations of GJ [18] and Pizza [19]. For example, the `Forward` class below has a single static method which can forward any type of data to a remote location. The particular type of data is chosen by the programmer and parameterises the call made.

```
class Forward() {
  static method forward[class packet] (
    dest : location,
    p : packet
  ) {
    if (@dest) {
      // do nothing
    } else {
      spawn towards dest {
        Forward.forward[packet]
          (dest,p);
      }
    }
  }
}
```

The same class can now be used to forward video packets, audio packets, and any other traffic. If we have a `Video` packet class defined, then we call

```
Forward.forward[Video]
            (dest,myVideoPacket);
```

to forward a single video packet. Note that unlike C++ templates, this class does not need to be instantiated separately for each packet type; instead each instantiation can share code and static data. By removing the need for casting, we have reduced a number of the run-time type errors which Java programs are prone to.

2.3 Located Objects

Most objects in a distributed program are only referenced locally, and are never referred to by remote sites. But some objects act as 'well-known resources', which can be referred to anywhere on the network. Such resources correspond to sockets, services, object registries, object databases, and so on. In SafetyNet, such resources are *located objects*. We have implemented a `located` class using the *visitor* design pattern, with the following class definitions:

```
native final serializable val class located
  [class Resource](init : Resource) {
   method accept(locatedVisitor[Resource]);
}

extensible val class locatedVisitor
  [class Resource]() {
  abstract method visitHere(Resource);
  abstract method visitRemote(location);
  method visitError() { }
}
```

To advertise a heap-allocated object outside of the location in which it resides, a located object is created, by a `located` object parameterised on the correct type and initialised with a reference to the object. For instance, if we have an `HttpServer` object, we use the following code:

```
let server : HttpServer = new HttpServer();
let serverAdvert : located[HttpServer]
   = new located(server);
```

The `serverAdvert` can now escape into the network. To use the server through this located reference, the programmer must create a `locatedVisitor` object which is passed into the located `accept` method. For the programmers' convenience we have dressed this up with syntax sugar resembling a switch statement eg.

```
...
switch(serverAdvert) {
   case Here(server : HttpServer) {
      // use server...
   }
   case Remote(loc : location) {
      spawn towards loc {
         // recursive call at new location
      }
   }
}
...
```

The switch statement is compiled into the creation of a `locatedVisitor[HttpServer]` object, which is then passed into the `accept` method of the located object. If the call is at the location of the server, then the `visitHere` method of the `locatedVisitor` object is called, otherwise the `visitRemote` method is called. This visitor pattern ensures that objects are never referenced outside of the location of their heap. Note that the `locatedVisitor` has a non-abstract visitError method which is empty. Programmers can override this method to provide error handling, rather than dying silently.

2.4 Types in Networked Programs

We now turn to the question of what types may be captured and sent in spawn expressions. In most object-oriented languages, objects are heap allocated, so copying an object to a new location either requires heap references to be valid across locations or a deep clone to be made. Both of these alternatives are costly, the former since it requires all objects to be externally registered, and the latter since it requires an object cache of all serialized objects.

Our solution has been to split objects into ones which are heap allocated, and ones which are stack allocated. For example, all of the primitive datatypes such as `int` and `double` are treated as stack-allocated objects, whereas more complex structures such as hash tables are heap-allocated. We can then require all serializable data to be stack-allocated, and to only contain stack-allocated fields. Again this is checked at compile time through the use of the type system. Classes are either stack allocated, indicated by the keyword `val`, or heap allocated, indicated by the keyword `ref`. In addition, if a `val` class is intended to be sent between locations, then it must be decorated with the `serializable` keyword.

For example we can declare:

```
serializable val class Complex (
  x : double, y : double
) {
  field x : double = x;
  field y : double = y;
}

native ref class Hashtable
  [class Key,class Item]() {
  method contains(Key) : boolean;
  method put(Key,Item);
  method get(Key) : Item;
}
```

then the following is legal:

```
  let c : complex = new complex (1.0,2.0);
  spawn at route dest {
    ...do something with c...
  }
```

but the following is not:

```
  let h : HashTable[string,location]
    = new HashTable[string,location] ();
  spawn at route dest {
    ...do something with h...
  }
```

We also do not allow any assignment to stack-allocated objects, so we ban cases such as:

```
  let x : int = 99;
  spawn at route dest { x := x + 1; }
```

Note from above that the `located` class is a serializable val class, so it can be transmitted over the network.

By using stack-allocation for serializable data, we can efficiently serialize and transmit data, and we make explicit the object cloning and caching that is implicit in Java RMI and other object serialization architectures. In addition, garbage collection is made much simpler and more efficient, since most data is now explicitly stack allocated, rather than heap allocated as in Java.

If we are going to allow classes to be serializable, then this means we have to decorate our generic classes to say which class parameters are going to be serialized:

```
  class OutputStream [class Data] {
    abstract method send (Data);
  }
  class Forwarder {
    static method forward [
      serializable class Packet
    ] (
      dest : location,
      os : OutputStream [Packet] at dest,
      p : Packet
    ) {
      if (at dest) {
        os.send (p);
      } else {
        spawn at route dest {
          Forwarder.forward
            [Packet] (dest,os,p);
        }
      }
    }
  }
```

The `OutputStream` class is used for communication within one location, and so does not need its data to be serializable. The `Forwarder` class, on the other hand, sends packets out over the network, so they need to be serializable.

2.5 A Simple RPC Mechanism

Class loading is dynamic as in Java. The semantics of class loading is to only load a class when it is actually used so that the transmission of records across multiple locations does not require a class to be loaded. For instance, in the RPC code in Figure 1, the `Function` class is loaded at the source and is then not loaded until the call reaches its destination.

The data which is sent between locations is:

- Header information identifying the packet as SafetyNet active code.
- A small code section requesting a call to `rpc.request` on the way out and `rpc.respond` ont he way back.
- Data payload consisting of the destination, the argument on the way out, and the result on the way back, the located heap reference to the Latch to return the value on, and the field values of the object.

Once the `rpc` class is loaded at all of the intermediate nodes along a route, the bandwidth overhead of using the active router rather than a conventional router is small: an identifying header, plus the method name to call, for a large gain in flexibility.

Other things of note in this example is the use of the `Latch` class to implement a set and blocking get on some data. This is declared native and uses internal mechanisms within the runtime to do the appropriate synchronization. We also extend the base RPC class from the `function` class, so that any RPC class can be called in the same fashion as `f.apply(arg)`. The destination is passed as a parameter when the RPC object is instantiated.

2.6 Linear types

Throughout the SafetyNet project, we are making use of types to guarantee safety features. Traditionally, this has meant, for example, using types to ensure that all memory access is safe. We hope to generalize this to show more subtle run-time behaviours such as resource usage, web-of-trust, and security. In this section, we shall show an example of using types to bound resource usage.

Bounding resource usage is a general problem. Within the Internet, the IP packet uses a *TimeToLive* field in the packet header to ensure that packet do not circulate for ever. Each time the packet passes through a router, the field is decremented, until it reaches zero, when the packet is thrown away. However, this does not give protection against programmers who maliciously or inadvertently create exponential packet forwarding loops such as in Figure 2.6.

We have generalised the *TimeToLive* field to be a general resource usage counter, which we have termed a *BeanCount*. The BeanCount type is derived from a theory of types known as *linear types*[20, 21], in which the values can only be used once. The BeanCount can be viewed as a protected integer. Each thread has an associated BeanCount value derived from the spawning. If each spawn operation ensures that the combined values of the BeanCount for the old and new threads is strictly less than the old thread value, then we can be sure that a thread cannot spawn threads indefinitely. We use syntactic sugar to hide the spawn count from most applications. However, we have provided a native `BeanTub` object, into which threads can dump beans or gather beans. This allows programs to implement useful functionality such as multicast.

2.7 Protyping in Java

Our prototype compiler and runtime environment has been built in Java, and the compiler compiles

```
native ref class Latch [class Data]() {
  method := (Data);
  method * () : Data;
}
extensible serializable val class function
  [class Argument, class Result]() {
  abstract method apply (Argument) : Result;
}
serializable val class rpc [
  serializable class Argument,
  serializable class Result
] ( f : function [Argument, Result],
    dest : location
  ) extends function [Argument, Result]() {
    field f : function [Argument, Result] = f;
    field dest : location = dest;
  method apply (a : Argument) : Result {
      let src : location = localhost;
      let fin : Latch[Result]
        = new Latch[Result]();
      let l : located[Latch[Result]]
         = new located[Latch[Result]](fin);
      spawn { this.request (a,src,l); }
      return *fin;
  }
  method request ( a : Argument,
    src : location,
    l : located[Latch[Result]]
  ) {
    let dst : location = this.dest;
    let f : rpc[Argument,Result] = this;
    if (@dst) {
      let r : Result = this.f.apply (a);
      this.respond (r, src, l);
    } else {
      spawn towards dst {
        f.request (a,src,l);
      }
    }
  }
  method respond (r : Result, src : location,
                  l : located[Latch[Result]]) {
    let f : rpc[Argument,Result] = this;
    switch(l) {
      case Here(fin : Latch[Result]) {
        fin := r;
      }
      case Remote(d : location) {
         spawn towards src {
           f.respond (r,src,l);
         } } } }
}
```

Figure 1: RPC code

```
serializable class Attack {
  method attack (
    src : location, dest : location
  ) {
    spawn at route dest {
      this.attack (src, dest);
      this.attack (dest, src);
    }
  }
}
```

Figure 2: Exponential looping code

into Java. This has allowed us to develop a working system quickly as we have evolved the language. However, we must emphasise that this is a prototype, since compiling into Java removes many of the expected performance benefits, and many of our safety features can be circumvented by producing Java classes that use the runtime API. Accordingly, we have not yet devoted much effort to tuning the network performance of the implementation, since this effort would be thrown away as we move to the SafetyNet Virtual Machine.

Since we are using Java as the base implementation, we can only approximately evaluate how the threads and records in SafetyNet map onto packets. We currently map each thread onto an anonymous class subclassed from an abstract Runnable SafetyNet thread we have named Snakes. The Snakes maintain information about the thread, such as its current BeanCount and the program identifier. The anonymous class mechanisms in Java already use closures, so we piggyback on this to create the class for the given thread by putting the thread code in the *run* method of the class. This can then be serialized into a byte array and sent as a packet. The current implementation uses a runtime check to ensure that the byte array is small enough for a single 1500 byte UDP packet. If the byte array is too big, it relies on the underlying IP fragmentation to send the packet. Since we do not guarantee the execution of threads, this is currently good enough.

An unfortunate side effect of Java serialization is that we unpack every packet into its component classes, even though parts of the data are not necessarily used, as in the RPC example. When we move to the SafetyNet Virtual Machine, we will be implementing lazy class loading, which only loads classes on demand.

We make use of dynamic class loader technology to build our own class loader using SafetyNet as the class directory. This has been remarkably easy to do, due to the excellent design of the class loading mechanisms in Java [22].

The use of stack allocated records does not necessarily imply that the implementation should allocate the records on the stack. In our current implementation in Java, the records map to Serializable classes, which are obviously heap allocated. In future implementations, we may make use of region-based analysis [23].

Our language is designed to be independent of the routing structure underneath. Deployment of SafetyNet nodes could use an "Active Network" bit to reroute the packet into the common computational engine of some switch, so using the existing routing infra-structure, or the network could be formed as an overlay network, such as the MBone or the putative ABone. In the current prototype implementation, routing is static and controllable, allowing testing. We are exploring using dynamic routing for future releases.

There are a number of support classes, such as timers and debugging channels. To allow configuration of topologies according to the current load and available bandwidth in the network, there is a measurement class which allows querying of the available bandwidth and other measurement statistics on the next hop to some host.

We have been able to use the known semantics of the language to allow some simple optimisation of communication patterns. We have built a number of RPC classes, based on the code shown in Figure 1.

The semantics of this code can be reduced to show that the function *Function.apply* will only ever be required on the *destination*, and that the *Argument* and *Result* are only passively carried across the intervening network without evaluation. This allows us to optimise the implementation of *RPC* so that the *Argument* and *Result* are transmitted directly across the network, rather than evaluate the thread code at each node. This approach to optimisation is an ad hoc use of partial evaluation based on the language semantics, which we hope to formalise and automate in future work.

3 Current Status

The language has now passed through three major revisions since the initial prototype compiler, changing the syntax and semantics of located objects, and of stack versus heap allocation. The compiler is now relatively stable and is in use for student projects and in teaching distributed systems. The current release can be found at `http://www.cogs.susx.ac.uk/projects/safetynet/`.

We believe that a key area of application for active services and networks will be in matching group communication to application requirements. We can match delivery mechanisms to the expected message and receiver populations, using variations on flooding for dense populations, whilst using low maintenance shared trees for long lived sparse groups. When information needs to be synchronised in some way across a group, the latency of the synchronisation can be much reduced if the synchronisation happens at the centre of the group rather than going to each node at the edge. For asynchronous communication, the program can act as the information repository, eg a migrating web server could follow time zones with a limited cache.

For synchronous delivery of messages to a group, we have generalised multicast. Since we have a polymorphic language, we have defined a base multicast class and an associated group identifier class, from which all implementations of multicast inherit. A student within our group is building a suite of multicast protocols withina common framework for particular applciation requirements, such as total ordered delivery, reliable multicast etc. Other students have developed applications such as an network intrusion detection system, a distributed virtual environment integrating the Java3D VRML browser with SafetyNet for distributing events, and a distributed wargame, based on the same principles as distributed simulation.

4 Future Work

For most applications, the packet forwarding code will simply receive a packet, update some state and send a packet onwards. We would like to be able to distinguish between fast path code, from control code, and place the fast path code in a special scheduling category. However, we need to be sure that the fast path code will terminate, and doesn't perform expensive operations such as heap allocation. To do this we will be using an extension to the type system based on *Categories of Computation*[24, 25, 26, 27]. We will define a sub-language of SafetyNet which doesn't have recursion (so guaranteeing termination), heap allocation or other expensive operators. Code which is typed as *Packet-forwarding* will be checked to ensure that it only uses the restricted sub-language and classes which are already type-checked as *packet-forwarding*.

Much of the work here has been inspired by the π-calculus [28], and builds on the work in which channels have been generalised to locations in [29, 30, 31]. These calculi have not only modelled communication networks; they have also been used to model security processes [32, 33, 34]. We will be using the nonces created in nominal calculi to model the chains of trust formed from principals to their code, and in the use of secure certificates. Other language work includes the use of partial evaluation to generate optimisations on the communication patterns within the code, and in using region based memory allocation.

Further work is planned to design and implement our own virtual machine and related code and data representation, so we can be sure that the safety features of the language are reflected in the machine, and we can obtain valid performance measures unencumbered by the use of the JVM as our runtime. The current prototype implementation is completely unsuited for performance evaluation, since we by necessity are using some very heavyweight features of the JVM such as serialization.

References

[1] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 1995.

[2] David D. Clark, Scott Shenker, and Lixia Zhang. Supporting real- time applications in an integrated services packet network architecture and mechanism. In *Proceedings of SIGCOMM 92*, 1992.

[3] David D. Clark and Wenjia Fang. Explicit allocation of best effort packet delivery service. http://diffserv.lcs.mit.edu/.

[4] L. Zhang et al. Rsvp: A new resource reservation protocol. *IEEE Network*, September 1993.

[5] Aurel Lazar, Koon-Seng Lim, and Franco Marconcini. Realizing a foundation for programmability of atm networks with the binding architecture. *IEEE Journal on Selcted Areas in Communications*, September 1996.

[6] Elan Amir, Steven McCanne, and Randy Katz. An active service framework and its application to real-time multimedia transcoding. In *Proceedings of SIGCOMM 98*, Vancouver, British Columbia, September 1998.

[7] Danny Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[8] Atanu Ghosh and Mike Fry. Application level active networking. In *Hipparch Workshop*. UCL, London, UK, June 1998.

[9] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 1996.

[10] Philippe Bernadat, Dan Lambright, and Franco Travostino. Towards a resource-safe java for service guarantees in uncooperative environments. In *IEEE workshop on Programming Languages for Real-Time Industrial Applications*, September 1998.

[11] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the spin operating system,. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–28, Copper Mountain, CO, 1996.

[12] Scott Alexander, Scott Nettles, Marianne Shaw, and Jonathan Smith. Active bridging. In *Proceedings of SIGCOMM 97*, Cannes, France, August 1997. ACM.

[13] Malcolm McIlhagga Ian Wakeman and Andy Ormsby. Signalling in a component based world. In *Proceedings of the First IEEE Open Architectures for Signalling*, San Francisco, Ca., April 1998.

[14] Malcolm McIlhagga, Ann Light, and Ian Wakeman. Towards a design methodology for adaptive applications. In *Fourth ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'98)*, October 1998.

[15] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, January 1996.

[16] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The switchware active network architecture. *EEE Network Special Issue on Active and Controllable Networks*, 12(3):29–36, 1998.

[17] A.S.A. Jeffrey. A distributed object caculus. Submitted to HOOTS 99., 1999.

[18] Gilad Bracha, Martin Odersky, David Stoutamire, , and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *OOPSLA 98*, Vancouver, October 1998.

[19] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *24th ACM Symposium on Principles of Programming Languages*, Paris, , January 1997.

[20] J.-Y. Girard. Linear logic. *Theoret. Comput. Sci.*, 50:1–102, 1987.

[21] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In *Proc. Int. Conf. Typed Lambda Calculi and Applications*, number 664 in LNCS. Springer-Verlag, 1993.

[22] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *OOPSLA 98*, Vancouver BC, October 1998. ACM.

[23] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[24] E. Moggi. Notions of computation and monad. *Inform. and Comput.*, 93:55–92, 1991.

[25] Philip Wadler. Comprehending monads. In *Proc. 1990 ACM Conf. Lisp and Functional Programming*, pages 61–78, 1990.

[26] A.J. Power and E.P. Robinson. Premonoidal categories and notions of computation. *Math. Struct. in Comput. Science*, 1999.

[27] Peter Selinger. Control categories: an axiomatic approach to the semantics of control in functional languages. Presented at MFPS '98, London, 1998.

[28] R. Milner, J. Parrow, and D. Walker. A calculus of mobile proceses. *Inform. and Comput.*, 100(1):1–77, 1992.

[29] Luca Cardelli and Andrew D. Gordon. Mobile ambients. Unpublished manuscript, 1997.

[30] Peter Sewell. Global/local subtyping for a distributed π-calculus. Technical Report 435, Computer Laboratory, University of Cambridge, 1997.

[31] J. Riely and M. Hennessy. Distributed processes and location failures. Technical Report 2/97, Univ. Sussex, 1997. Also presented at ICALP '97.

[32] A. Gordon and M. Abadi. A calculus for cryptographic protocols: The spi calculus. Research Report 149, Digital Equipment Corporation Systems Research Center, 1998. To appear in *Information and Computation*.

[33] Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. ACM Conf. on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[34] Andrew D. Gordon. Nominal calculi for security and mobility. In *Proc. DARPA Workshop on Foundations for Secure Mobile Code*, pages 10–14, 1997.