Extending Erlang for Safe Mobile Code Execution

Lawrie Brown*, Dan Sahlin†

*School of Computer Science, Australian Defence Force Academy, Canberra 2600, Australia
†Computer Science Laboratory, Ericsson Telecom, S-126 25 Stockholm, Sweden

November 5, 1997

Abstract

This paper discusses extensions to the functional language Erlang which provide a secure execution environment for remotely sourced code. This is in contrast to much existing work which has focused on securing procedural languages. Using a language such as Erlang provides a high degree of inherent run-time safety, which means effort can be focused on providing a suitable degree of system safety. We found that the main changes needed were the use of unforgeable (capability) references with access rights to control the use of system resources; the provision of a hierarchy of execution nodes to provide custom views of the resources available and to impose utilisation limits; and support for remote module loading. We then discuss prototype implementations of these changes, used to evaluate their utility and impact on visibility for the users of the language, and mention work in progress using this foundation to specify safety policies by filtering messages to server processes.

1 Introduction

We present some extensions to the functional language Erlang which provide a secure execution environment for mobile code. Because the code is sourced remotely, with a lower level of trust than local code, it needs to be executed within some form of *constrained* or *sandbox* environment to protect the local system from accidental or deliberate inappropriate behaviour. A key assumption is that the local system is trusted, and provides adequate access control mechanisms. We also assume that appropriate protection of inter-node communications is available, for example by IPSEC [5], SSL [15] or similar mechanisms.

In his taxonomy of issues related to distributed hypermedia applications, Connolly [11] in particular distinguishes between *Run-time Safety* — which deals with whether the runtime system will guarantee that the program will behave according to its source code (or abort); and *System Safety* — which deals with whether it is possible to control access to resources by a piece of executing code. A similar emphasis on constraining resource access is identified by Banks [6] and Adl-Tabatabai et al. [1] amongst others.

Much existing work on providing safe execution environments has focused on securing procedural languages and their run-time environment, including Java [14], SafeTCL [20], Omniware [18, 1], and Telescript [22], amongst other systems (see overview by Brown [8]). With these languages, much effort has to

*Phone: +61 2 62688184, Email: Lawrie.Brown@adfa.oz.au
†Phone: +46 8 7198187, Email: dan@cslab.ericsson.se

be expended to provide a suitable degree of run-time safety in order that the system safety can then be provided. A number of attacks on these systems have exploited failures in run-time safety (cf Banks [6], Dean et al. [13], or Yellin [24]).

In contrast, a dynamically typed, functional language can provide a very high degree of intrinsic run-time safety, since changing the type interpretation should be impossible (except perhaps via explicit system calls). This is noted by Connolly [12], who observes that languages like Objective ML and Scheme48 provide a high degree of run-time safety, though they need further work to provide an appropriate level of system safety. A fundamental assumption here is that the basic run-time system correctly enforces these type accesses, though the language semantics make checking and verifying this considerably easier.

1.1 Erlang

For the work discussed in this paper, the authors, initially independently, decided to concentrate on the language Erlang [4, 3, 21, 23]. Erlang is a dynamically typed, single assignment language which uses pattern matching for variable binding and function selection, which has inherent support for lightweight concurrent and distributed processes, and has error detection and recovery mechanisms. It was developed at the Ericsson Computer Science Laboratory. Most of the Erlang system is written in Erlang (including compilers, debuggers, standard libraries), with just the core run-time system and a number of low-level system calls (known as BIFs for BuiltIn Functions) written in C. Distributed use is almost transparent, with processes being spawned on other nodes, instead of locally. An Erlang node is one instance of the run-time environment, often implemented as a single process (with many threads) on Unix systems. Erlang is currently being used in the development of a number of very large telecommunications applications, and this usage is increasing. It is expected that applications development will be increasingly outsourced, but that they will be executed on critical systems. It is also anticipated that there will be a need to support applications which use mobile agents, which can roam over a number of systems. Both of these require provision of an appropriate level of system safety. It was to satisfy these identified requirements that the authors decided to evaluate and prototype appropriate extensions to Erlang to provide such system safety. Because the language is in production use, there is a requirement to minimise the visibility of any changes to the users, except when explicitly using the new features.

2 Current Limitations

Using Erlang as the core language for a safe execution environment provides a number of advantages. For instance, *pure* functional language use, where functions manipulate dynamically typed data, and return a value, is safe (apart from problems of excessive utilisation of cpu, memory etc). It is only when these functions are permitted to have side-effects that they may threaten system safety. Side effects are possible in Erlang when a function:

accesses other processes by sending and receiving messages or signals, viewing process information, or changing its flags.

accesses external resources outside the Erlang system (files, network, hardware devices etc), using the `open_port` BIF.

accesses permanent data in databases managed by the database BIFs.

Thus, a safer Erlang requires controls on when such side-effects are permitted. Consider the following code excerpt from an account management server, which when started, registers itself under the name `bank`, and then awaits transactions to update the account balance:

```
-module(bankserver).
-export([start/1]).
start(Sum) ->                  % start account server
    register(bank,self()),     % register as 'bank'
    account(Sum).              % process transactions
account(Sum) ->                % transaction processing loop
    receive                    % await transaction
      {Pid, Ref, Amount} ->    % account update msg received
        NewSum = Sum+Amount,   % update sum
        Pid ! {Ref,NewSum},    % send response back
        account(NewSum);       % loop (recurse)
      stop -> nil              % end server msg received
    end.
```

This could be started with a balance of 1000, and updated, as follows:

```
...
% spawn a bank account process with initial Sum 1000
Bank_pid = spawn(BankNode,bankserver,start,[1000]),
...
Ref = make_ref(),              % make a unique ref value
Bank_pid ! {self(),Ref,17},    % send msg to server
receive                        % await reply from server
    {Ref,New_balance} ->       % reply says updated ok
        ...
end,
...
```

In standard Erlang the data items used to access processes and external ports (pids and ports) are both forgeable and too powerful. Apart from legitimately obtaining a pid by being its creator (eg `Bank_pid = spawn(BankNode,bankserver,account,[1000])` in the example above), receiving it in a message (`receive Bank_pid -> ok end,`), or looking it up via a registered name (`Bank_pid = whereis(bank)`); it is also possible to obtain a list of all valid pids on the system (`AllPids = processes()`), or to simply construct any arbitrary pid from a list of integers (`FakePid = list_to_pid([0,23,0])`). These latter features are included to support debugging and other services, but open a significant safety hole. Further, once a valid pid has been obtained, it may be used not only to send messages to the referenced process (`FakePid!{self(),Ref,-1000}`), but to send it signals, including killing it (`exit(FakePid,kill)`), inspect its process dictionary (`process_info(FakePid)`), change its priority (`process_flag(priority,low)`) etc. And there is no mechanism, currently, to limit the usage of a pid (to just sending messages, for example).

Another limitation of the current Erlang system from a safety perspective is the fact that a given Erlang system (that is, once instance of the run-time environment) forms a single node. All its processes have the same privileges and the same access to all resources (file system, modules, window manager, devices) managed by the system. There is no mechanism to partition access within a system, so that it may be mediated via a trusted server process. The only current method for providing this is to run a separate system, at a considerable cost in system (cpu, memory etc) resources. There is also no means to limit the resources utilised by processes, apart from imposing restrictions on the entire system.

Lastly, there is a need to provide a remote module loading mechanism in order to properly support mobile agents. Whilst Erlang currently supports distributed execution and remote spawning of processes, the module containing the code to be executed must exist on the remote system in its default search path. Further, any modules referenced in the loaded module will also be resolved on the remote system. The code loading must, however, be implemented in such a way that the remote code server cannot be tricked into sending code that is secret. Note that pid and port identifiers are globally unique, so they may be passed in messages between nodes whilst maintaining their correct referent.

3 Extensions for a Safer Erlang

In order to address the deficiencies identified above, we believe that extensions are required in the following areas:

capabilities may be used to impose a finer granularity of control on the use of process (and other resource) identifiers, making these unforgeable with a specified set of rights on the use of any specific instance.

nodes that form a hierarchy within an Erlang system (one instance of the runtime environment), to provide a *custom context* of services available, to restrict the use of code with access to external resources, and to impose utilisation limits (on cpu, memory etc).

remote module loading may be used to allow spawning of modules sourced on another system, retaining knowledge of the source system so that subsequent module references can also be sourced from that system.

These extensions are described in more detail below.

3.1 Capabilities

We define a capability as a globally unique (in time and space), unforgeable name for (a specific instance of) some resource and a list of rights for operations permitted on the resource by holders of the capability.

The same resource may be referred to from different capabilities giving the owners of the capabilities different rights. We are using capabilities to ensure that these identifiers cannot be forged, and to limit their usage to the operations desired. A capability is always created (and verified upon use) on the node which manages the resource which it references, and these resources never migrate. This node thus specifies the *domain* for the capability. Further, the resources referenced are never reused (a new process, even with the same arguments, is still a new instance, for example), so revocation is not the major issue it traditionally is in capability systems. In our usage capabilities are invalidated when their associated resource is destroyed (eg. the process dies). Other processes may possess invalid capabilities, but any use will raise an *invalid capability* exception.

Capabilities may be implemented in several ways.

Encrypted (hash) Capabilities use a cryptographic hash function (eg [17]) to create a encrypted check value for the capability information, which is then kept in the clear. Only the node of creation for the capability can create and check this hash value. The overhead of validating the check value can be minimised if any *local* encrypted capabilities are checked once on creation (or import) and then simply flagged as such (say by amending the hidden data type tag to indicate that it has been verified). Subsequent use of the capability then incurs no overhead. Further, for *remote* capabilities, any delays due to cryptographic overheads are likely to be swamped by network latencies. Each node would keep the key used to create and validate its capabilities secret, and this key could be randomly selected when the node is initialised. Any previously created capabilities must refer to no longer extent (instances of) resources (from a previous incarnation of the node), so there is no requirement to continue to be able to validate them. This approach could be attacked by attempts to guess the key used, and verifying the guess against intercepted capability data. The likelyhood of success will depend on the type of hash function used, so some care is needed in its selection to avoid known flaws in some *obvious* modes of use [7].

Password (sparse) Capabilities [2] use a large random value (selected sparsely from a large address space) to protect the capability, with the node of creation maintaining a table of all its valid capabilities. This table is checked whenever it is presented with a capability, and capabilities may be revoked by removal from this table. One disadvantage of this approach is the size this table may grow to, particularly for long running server processes, or when user defined capabilities are used, where it is impossible to know when they have no further use. Another is that large tables may take some time to search, though careful selection of the table mechanism can reduce this to a minimum. It is possible to try and forge such a capability, but success is statistically highly improbable, and attempts should be detectable by abnormally high numbers of requests presenting invalid capabilities.

There is thus a tradeoff between these alternatives — trading some level of security with encrypted capabilities for space with password capabilities. The best alternative is likely to depend on the target environment.

Experience with the prototypes has shown that it is important for efficient execution that all information needed to evaluate guard tests or pattern matches be present locally in the capability. This information must include the type (for the various mid/nid/pid/port etc guards) and the value (to test if two capabilities refer to the same object). Because different applications may wish to choose between the security tradeoffs, we decided to support both hash and password capabilities, chosen on a node by node basis, in an interoperable mechanism, where only the node of creation need know the actual implementation. Thus, we have chosen to use capabilities with the following components: `<Type,NodeId,Value,Rights,Private>`, where:

Type the type of resource referenced, eg. module, node, pid, port, or user.

NodeId the node of creation, which can verify the validity of the capability or perform operations on the specified resource.

Value the resource instance referenced by the capability (module, node, process identifier, port identifier, or any Erlang term, respectively)

Rights a list of operations permitted on the resource. The actual rights depend on the type of the capability. For a process capability these could include: `[exit,link,register,restrict,send]`.

Private an opaque term used by the node of creation to verify the validity of the capability. It could either be a cryptographic check value, or a random password value: only the originating node need know.

A capability may be restricted (assuming it permits it). This results in the creation of a new capability, referencing the same resource, but with a more restricted set of rights. Using this, a server process can, for example, register its name against a restricted capability for itself, permitting other processes to only send to it. eg. `register(bank, restrict(self(),[send,register]))`

Capabilities would be returned by or used instead of the existing node names, pids, or ports, by BIFs which create or use these resources.

3.2 Nodes

A hierarchy of **nodes** are provided within an Erlang run-time system (an existing Erlang **node**). These would provide a *custom context* of services available, restrict the use of code with side-effects, and impose utilisation limits (eg on cpu, memory etc). Functionally, these would be similar to existing Erlang **nodes** with some additional features.

A **custom context** for processes is provided by having distinct:

registered names table of *local* names and associated capability identifiers, used to advertise services. These names are not visible on other nodes.

module alias table which maps the name used in the executing code to the name used locally for the loaded module. This module name *aliasing* mechanism is used to support both redirection of names to *safer* variants of modules, as well as to provide unique names for modules loaded from remote systems (for agents or applets). This table is consulted to map the module name on all external function calls, applys, and spawns.

capability data which is all the information (eg. keys or table) necessary to create and validate unforgeable capabilities for the node.

Restrictions on **side-effects** are enforced by specifying whether or not each of the following are permissible for all processes which execute on the node:

- **open_port** for direct access to external resources managed by the local system.
- **external process access** for access to processes running on other Erlang systems, which could provide unmediated access to other resources, or reveal information about the local system to other nodes.
- **database BIFs usage** for access to permanent data controlled by the local database manager.

When disabled, access to such resources would have to be mediated by server processes running on the local system, but in a more privileged node, trusted to enforce an appropriate access policy for safety. Typically these servers are advertised in the registered names table of the *restricted* node.

Utilisation limits can be imposed by a node for all processes executing within the node, or any descendent child nodes of it. Limits could be imposed on some of cpu usage, memory usage, max no reductions; or perhaps on combinations of these.

The general approach to creating a controlled execution environment is as follows. First, a number of servers are started in a node with suitable privileges to provide controlled access to resources. Then a node would be created with *side-effects* disabled. Its *registered names* table would be pre-loaded to reference these servers, its *loaded modules* table pre-loaded with appropriate local library and safe alias names; and appropriate utilisation limits set. Processes would then be spawned in this node to execute the desired *mobile code* modules, in a now appropriately constrained environment. eg. `BankNode=newnode(node(),ourbank,[{proc_rights,[]}])`

3.3 Remote Module Loading

Our system does not directly support mobile processes as once started, a process may not migrate. However, a new process may be spawned off at a remote node. Extensions are therefore needed so that subsequent module references from within a remotely loaded module can also be sourced from the same node. This involves an extension of the `apply` BIF handler so that it checks whether the originating module is local or remote, and proceeds accordingly to interpret the module requested in context, querying the code server on the remote node for the module, if necessary. Some care is needed in managing the acquisition of an appropriate capability for requested module. This is issued by the remote code server upon receipt of a request which includes a valid capability for the parent module, and is then used to request the loading of the requested module.

4 Prototypes

A number of prototypes of a safer Erlang environment have been trialed.

In Uppsala, as a students project, a design for safe mobile agents, was implemented in 1996 [16]. Distributed Erlang was not used in that system, which instead was based on KQML communication. Safety was supported by protected *secure domains* spanning a number of nodes, where it was assumed that all nodes within a single domain were friendly.

SafeErlang was developed by Gustaf Naeser and Dan Sahlin in during 1996 [19]. The system supports a hierarchy of subnodes to control resource usage and to support remotely loaded modules; *encrypted capabilities* for pids and nodes to control their usage; and remote module loading. Whilst this prototype was successfully used to trial a mobile agent application, limitations were found with the complexity of its implementation, the incomplete usage of capabilities for all resource items (in particular for ports), and the use of fully encrypted capabilities and the consequent need to decrypt them before any information could be used.

More recently the *SSErl* prototype was developed by Brown [10] whilst on his sabbatical in 1997 at SERC [1] and NTNU[2]. It supports a hierarchy of nodes on each Erlang system which provide a *custom context* for processes in them, the use of both *hash* and *password capabilities* for pids, ports, and nodes to constrain the use of these values; and remote module loading. This prototype has evolved through a number of versions in the process of refining and clarifying the proposed changes.

Both prototypes implement the language extensions using glue functions for a number of critical BIFs. These are substituted by a modified Erlang compiler (which itself is written in Erlang). The glue routines interact with *node* server processes, one for each distinct node on the Erlang system. Most of the SSErl glue functions have the form:

[1]Software Engineering Research Centre, RMIT, Melbourne, Australia
[2]Norwegian University of Science and Technology, Trondheim, Norway

- possibly resolve a registered name to capability
- check with the node manager to see if the operation is permitted by the capabilities rights, and if not, to throw an exception
- otherwise some key information is returned (eg a real pid or capability)
- perform the desired operation (in the user4s process)

For example, the `k_exit` glue routine look as follows:

```
k_exit(CPid,Reason) ->
    Pid = node_request(check,CPid,?exit), exit(Pid,Reason).
```

Both support a hierarchy of **nodes** within an Erlang run-time system (an existing Erlang **node**). Each node is represented by a node manager process, which manages the state for that node, and interacts with the glue routines to manage resource access.

SSErl capabilities are a tuple with the components identified previously: {`Type,NodeId,Value,Rights,Private`} which specify the type of resource the capability references, the node managing that resource, the resource instance, the list of access rights permitted on the resource, and the opaque validation value (crypto hash or password) for the capability.

To support these new features, SSErl provides some new BIFs:

check(Capa,Op) checks if the supplied capability is valid and permits the requested operation, throwing an exception if not. This is not a guard test as it must consult the originating node to validate the capabilities check value.

halt(Node) halt a node along with all its nodes and processes.

make_capa(Value) create a user capability with value given.

make_mid(Module) create a module (mid) capability for the named module.

newnode(Parent,Name,Opts) creates a new node as a child of the Parent, with the specified context options.

restrict(Capa,Rights) creates a new version of the supplied capability, referring to the same resource, but with a more restricted set of rights.

same(Capa1,Capa2) guard testing whether the supplied capabilities refer to the same resource, without verifying the check value (for efficiency reasons).

Except when explicitly configuring the execution environment, the new features are mostly invisible to existing user programs. The SSErl prototype successfully compiles many of the standard library modules (only those interacting with ports require some minor, systematic, changes necessitated by the protocol currently used). It is now being used to trial some demonstration applications.

In the future we anticipate that the extensions demonstrated here will be incorporated into the Erlang run-time environment. This should remove some unavoidable incompatibilities (such as with ports) found in the prototypes, as well close off any possibilities of bypassing these safety extensions.

5 Custom Safety Policies

The extensions described above provide mechanisms necessary to impose an appropriate safety policy, without enforcing any specific policy. Work is now continuing on providing an appropriate level of *system safety* by executing the mobile code in a constrained environment created using these extensions, trusted to prevent direct access to any external resources by the executing mobile code, and then applying filters to messages between the mobile code and server processes which mediate the access to resources. This allows the use of standard servers, but with distinct policies being imposed for each instance. By restricting the security problem domain to just verifying that the messages conform to the desired policy, the problem should be considerably more tractable than approaches based on proofing the imported code directly, for example. It is also much more efficient than approaches which attempt to validate all function calls. The use of such a safety check function on client-server messages is further elaborated on in Brown [9].

6 Conclusions

This paper describes proposed extensions to Erlang to enhance its ability to support safe execution of remotely sourced code. The changes include the provision of a hierarchy of nodes to provide a *custom context*, restrictions on *side-effects*, and resource limits; and the use of unforgeable references (capabilities) with associated rights of use for modules, nodes, processes, ports, and user defined references. Also, extensions are needed to support remote code loading in context. Two experimental prototypes, used to evaluate the utility of these extensions, are then described. Finally, mention is made of ongoing work using this safe execution environment as a foundation for imposing custom safety policies by filtering messages to server processes.

References

[1] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and Language Independent Mobile Programs. *SIGPLAN*, 31(5):127–136, May 1996.

[2] M. Anderson, R.D. Pose, and C.S. Wallace. A Password Capability System. *The Computer Journal*, 29(1):1–8, 1986.

[3] J. Armstrong. Erlang - A Survey of the Language and its Industrial Applications. In *INAP'96 - The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, Hino, Tokyo, Japan, Oct 1996. http://www.ericsson.se/cslab/erlang/publications/inap96.ps.

[4] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996. http://www.ericsson.se/erlang/sure/main/news/book.shtml.

[5] R. Atkinson. Security Architecture for the Internet Protocol. Technical Report RFC 1825, IETF, Aug 1995.

[6] J.A. Banks. Java Security. Technical report, MIT, Dec 1995. http://www-swiss.ai.mit.edu/~jbank/javapaper/javapaper.html.

[7] M. Bellare, R. Canetti, and H. Krawczyk. Keyed Hash Functions and Message Authentication. In *Advances in Cryptology - Proceedings of Crypto'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1996. http://www.research.ibm.com/security/keyed-md5.html.

[8] L. Brown. Mobile Code Security. In *AUUG 96 and Asia Pacific World Wide Web 2nd Joint Conference*. AUUG, Sept 1996. http://www.adfa.oz.au/~lpb/papers/mcode96.html.

[9] L. Brown. Custom Safety Policies in SSErl. Technical note, Australian Defence Force Academy, Canberra, Australia, Jun 1997. http://www.adfa.oz.au/~lpb/papers/ssp97/sserl97e.html.

[10] L. Brown. SSErl - Prototype of a Safer Erlang. Technical Report CS04/97, Australian Defence Force Academy, Canberra, Australia, Nov 1997. http://www.adfa.oz.au/~lpb/papers/tr9704.html.

[11] Dan Connolly. Issues in the Development of Distributed Hypermedia Applications, Dec 1996. http://www.w3.org/OOP/HyperMediaDev.

[12] Dan Connolly. Mobile Code, Dec 1996. http://www.w3.org/MobileCode/.

[13] D. Dean, E.W. Felten, and D.S. Wallach. Java Security: From Hotjava to Netscape and Beyond. In *Proceedings IEEE Symposium on Security and Privacy*. IEEE, May 1996. http://www.cs.princeton.edu/sip/pub/secure96.html.

[14] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems, May 1995. ftp://ftp.javasoft.com/docs/.

[15] T.J. Hudson and E.A. Young. SSLeay and SSLapps FAQ. Technical report, Uni. Queensland, 1996. http://www.psy.uq.edu.au:8080/~ftp/Crypto/.

[16] I. Jonsson, G. Naeser, D. Sahlin, and et al. Adapting Erlang for Secure Mobile Agents. In *Practical Applications of Intelligent Agents and Multi-Agents: PAAM'97*, London, UK, Apr 1997. http://www.ericsson.se/cslab/~dan/reports/paam97/final/paam97.ps.

[17] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. Rfc 2104, IETF, Feb 1997.

[18] S. Lucco, O. Sharp, and R. Wahbe. Omniware: A Universal Substrate for Mobile Code. In *Fourth International World Wide Web Conference*. MIT, Dec 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/165/.

[19] G. Naeser. Your First Introduction to Safeerlang. Technical report, CS, Uppsala University, Jan 1997. http://www.csd.uu.se/~gaffe/general/safe/nae97a.ps.gz.

[20] J.K. Ousterhout, J.Y. Levy, and B.B. Welch. The Safe-Tcl Security Model. Technical report, Sun Microsystems Laboratories, Mountain View, CA 94043-1100, USA, Nov 1996. http://www.sunlabs.com/research/tcl/safeTcl.ps.

[21] Dan Sahlin. The Concurrent Functional Programming Language Erlang - An Overview. In *Workshop on Multi-Paradigm Logic Programming, Joint Intl. Conf. and Symposium on Logic Programming*, Bonn, 1996. http://www.ericsson.se:800/cslab/~dan/reports/mplp/web/mplp.html.

[22] J. Tardo. An Introduction to Safety and Security in Telescript. Technical report, General Magic Inc., 1995. http://cnn.genmagic.com/Telescript/TDE/security.html.

[23] C. Wikstrom. Distributed Programming in Erlang. In *PASCO'94 - First International Symposium on Parallel Symbolic Computation*, Sep 1994. http://www.ericsson.se/cslab/erlang/publications/dist-erlang.ps.

[24] F. Yellin. Low Level Security in Java. In *Fourth International World Wide Web Conference.* MIT, Dec 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html.