Uppsala Master's Theses in
Computing Science 109
Examensarbete DV3
June 9, 1997
ISSN 1100–1836

SafeErlang

Gustaf Naeser

Computing Science Department
Uppsala University
Box 311
S-751 05 Uppsala
Sweden

Supervisor: Dan Sahlin
Examiner: Håkan Millroth
Passed:

Abstract

Erlang is a process oriented functional programming language developed for fast prototyping of soft real time distributed systems. The language could be suited for implementation of mobile agents if it supported the security which is needed in such systems. The code loading currently used in the language with its demand that code is locally available, needs also be modified to enable transparent mobile programming. This paper describes the design of extensions which make the language secure. They also enable the design of safer systems and also provide a new modified code loading.

A prototype has been implemented which shows that the extensions add the needed security. The prototype has also shown that the extensions with advantage can be used in other applications.

Contents

Chapter 1

Introduction

The demand for a safer and more secure ERLANG becomes apparent when the language is tried in new areas of programming. One such area is agent programming. Agents are programs devised to solve specific tasks using a defined behaviour. When the agents start moving around in a network or distributed system they are called mobile agents. The environments in which they execute are vulnerable and measures to make them safe and secure must be taken.

The foundation for mobile agent programming is that live executable code can be sent between clients in a network, rather than having the clients exchange dead data only. The networks considered in this paper are open. In an open system all components are not trusted so there are many possible security threats.

The security issues found in agent programming will in this paper be used to identify and then be the base for a design of extensions to the ERLANG programming language. The implemented prototype has also shown that even if the extensions were designed with agent security in mind, they are general enough to be used in other types of problems as well. They impose no severe limitations on the language and one goal has been that older programs with little or no modification should be able to run in the extended system. One other goal has been to transform ERLANG into a more secure language, rather than to create a secure environment inside ERLANG.

Related work

Most work concerning agents and agent programming has not focused on the security issues. Many agent systems and languages seem to regard security as a feature, not a requirement. Many systems stress the importance, but only suggests solutions rather then implementing them.

Some languages have implemented various levels of security and the main influences on the extension in this paper come from:

- Agent Tcl [Gra95] is a modified version of Tcl 7.5 [LO95]. It provides the commands agents use to communicate, migrate and create child agents. Authentication, based on PGP, is used to protect the machines form agents tampering. The agent's owner is identified and access permissions based on this authentication are assigned to the agent. Resource protection is maintained using Safe Tcl [Bor92, Bor94] and a set of resource manager agents.
- Telescript [Gen95, Whi94, Whi95] is developed by General Magic. The language an interpreted object oriented language with dynamic code loading. Runtime type checking, automatic memory management with garbage collection and exception processing makes scripts either succeed or fail gracefully. A mechanism using permits is used to set quotas for limiting resource usage and controlling the capabilities of objects. If an object has a number of permits, the engine computes the effective permit as a logical intersection of the applicable permits. Tardo and Valente [TV96] notes that with Telescripts mechanisms, safety can only be achieved by defensive coding practices.
- Java [Sun94, Sun95, Yel96] is an interpreted language with garbage collection. Although not a dedicated agent programming language, it is used in similar environments and contains several solutions to agent security issues. The definition of the language is strict and programs will execute identically regardless of the compiler or run-time system used. A extensive, stringent, compile-time checking tries to find as many errors as possible at compile time. The byte code is verified before it is passed to the interpreter by a verifier which ensures that a set of constraints hold, e.g. that code does not over- or underflow the stack or tries to use registers in invalid ways. What Java lacks is a way to manage resource consumption.

The structure of this report

The report is organised into the following chapters:

Chapter 2 introduces the concept of agents and describes the security requirements of agent environments.

Chapter 3 gives a brief introduction to the parts of ERLANG which needs to be modified by the extensions. Some other parts vital to agents programming are also described.

Chapter 4 describes the design of the proposed extensions.

Chapter 5 overview of the prototype.

Chapter 6 conclusions and further work.

Chapter 2

Agents

In later years *agents* have become a term more and more used. Every programming language and system with self respect supports them in some way. There are however one side of agents which has not been solved most systems, the security.

This chapter will introduce the concept of agents and different issues that will have to be addressed by a language which supports possibly malignant agents.

2.1 Definitions of agents

There are several different definitions on what agents are. Most of these definitions concern the behaviour of agents and different kinds of properties they should have. Woolridge and Jennings [WJ95] have proposed the behaviours autonomity, social ability, reactivity and pro activeness. These makes it possible to reason about the expected behaviour of an agent. One way of describing what agents are is therefore that an agent is a program with a specific task which makes it necessary for it to implement some or all of a set of properties, not necessarily the set mentioned above.

When agents start interacting with each other they form an *agent application.* It is possible that the application only has two agents where one of them is a simple interface to the user, but applications can have a much larger number of agents. An example of a two agent application is an intelligent mail watcher which sorts the mail in a way the user wants it. A multi agent application can be an electronic marketplace where agents buy and sell items and services.

When agents exist in a distributed system, they might be able to solve their task in a more efficient way, both for themselves and for the total efficiency of the distributed system, if they are allowed to move between the different hosts. Agents moving around, or travelling, in a system are called *mobile agents*. Harrison et. al. [HCK95] have discussed whether or not mobile agents can be used to improve the performance of an application. In general it can be said that only a few applications really enhance their performance when using agents, but that there are several other gains which can justify the use of agents. Applications can for example be made much easier to modify and extend if agents are used. As an example we can look at distributed queries. Assume that we have access to a huge database and want to collect data from it using a set of rules. There are three basic solutions to this problem. Either we can collect all data from the server and then locally process it to refine the data we want; we can let the server, given a query in some query language, give us the information we want; or we can send an agent to the server, and then let the agent collect the data we want. That the two latter cases are superior to the first should be obvious, but the difference between the query and agent cases is less distinct. The easiest way to see the difference is the action taken when the data has been collected and the work at the server is done. In the query case the data is simply returned to querier, but in the agent case a multitude of different scenarios are possible—the agent can act on the information it has collected and, should this be the case, need not even send all information back to the querier. It is possible that the agent finds out that it needs supplementary information and moves on to another site to find this information.

Even if this paper focuses most at mobile agents and the security issues concerning them, all agents do not have to, and will not, be mobile. There are several agents in a system which do not need to be mobile, e.g. database or file system.

2.2 Representation of agents

Lignau and Drobnik [LD95] has presented a model where agents themselves consist of three parts. The first part is their description, code, which defines the behaviour and task of the agent. The format of this description is dependent of the environment the agent executes in. The commonly used variants are that the agents are described in source code or scripts, or byte code. In terms of security the byte code representation seems to introduce security problems which can be hard to identify and handle without help from advanced tools. As an example, Java [Sun94] relies on that its loader, which installs foreign code into the system, does its job correctly. If on the other hand source code, or some representation of it like a parse tree, is used the trust is placed in the compiler instead. It is easier to create a compiler that cannot generate malignant code than a verifier which can detect if an arbitrary piece of code is malignant.

The second part of agents is their knowledge database. In some systems this database is called a *suitcase*. How the knowledge of an agent should be managed, and subsequently moved, can be handled in two ways:

- The runtime environment keeps track of all the information belonging to a specific agent. When the agent chooses to move, the environment gathers the information and transmits this together with the agent description.
- The agent itself is responsible for that the information is transmitted to the new location.

The third part of an agent is its attributes. Attributes are information about the agent, e.g. its place of origin, an unique identifier of the agent and possible resource requirements.

The anatomy of an agent can be various forms. In this paper an agent consists of one process. If the agent creates child processes these will be consedered to be thralls and not part of the agent. One reason for this view of an agent is that it makes it much easier to capture the agents state. The responsibility of managing the agent rests on the application programmer and not on the agent environment.

2.3 Environments for mobile agents

An environment which is to be the base for mobile agent programming will at least need some specific properties. These have discussed in several papers [LD95, HCK95, Whi95]. Attributes needed in an agent programming environment for mobile agents are

- communication;

- transport; and
- security.

2.3.1 Communication

Agents may have to use communication to solve, or better solve, their assignments. The communication is used to exchange and obtain information, negotiate or state purposes. Mostly agents may wish to communicate with other agents but the communication with its owner, to report results and or receive new tasks or directions, is also important. Standards for agent communication have been designed, KQML [FFM$^+$93] and KIF [GF92], in an effort to make agent systems able to communicate within themselves as well as with other agent systems.

There are two main ways in which communication can be managed in agent environments. Genesereth and Ketchpel [GK94] call them *direct communication* and *assisted coordination.*

In a system with direct communication, agents communicate directly with other agents. This does not rely on other programs than the agents. The alternative is to implement a *federated system* in which the agents communicate with *facilitators* which in turn communicate with agents or other facilitators. Communication in federated systems can more easily be controlled, screened or processed, but there will always be a slowdown due to the extra communication to and from the facilitators.

2.3.2 Transport

Means of transportation are needed in applications where mobile agents are used. The agents travel between the nodes in an distributed system in an effort to more efficiently solve their assignments. The choice when and where to move is made by the agents themselves.

When agents move there are three main things which need to be considered.

- How the agents knowledge is moved;
- how the agents description, code, is to be moved (and also if the state of the agent, i.e. the procedure stack, should be moved); and
- what the agent should execute upon arrival to the destination node.

Another problem is references to the agent. When an agent leaves a location, references to that agent might have to be updated. The references, if global references are not used, points to the agent on the location it is leaving and they should be updated to point to the agents new location. Related problems are issues like what happens with messages underway to the old location and how long a redirection, forwarding, from the old location need to be kept.

Moving agent knowledge

The solutions of how the knowledge is moved is closely related to how the knowledge is stored. If the environment manages the knowledge it should also be responsible for transmitting it to the agents new location. If the agent manages its own knowledge, it should be responsible.

A security problem is that agent knowledge could be modified while it is transported to a new location.

Moving agent descriptions

How the agents code is transmitted to the new location is a bigger problem than how its knowledge is moved. The way in which this is handled can not only put great limitations on programming style but it can also introduce security problems. Environments demanding that the modules that will have to be moved with the agent can be decided prior to the transfer of the agent, makes it harder for the programmer to create distributed applications. The mechanism for agent transportation is however made much easier.

There is also the question if the state of the agent need to be captured and transmitted. If the agent consists of one process only this is not a problem but if the agent consists of several processes the state of every one of these must be captured if the agent is to be started as an identical copy on the new place. This is why the agents in this paper are considered to consist of one process only.

As with the knowledge transportation, the agent description can be modified while being transported to a new location. A related problem is that there is no way for a agent to check that it actually executes the description it believes it is executing.

Restarting agents

There are different views on how the agents should arrive and be received by agent environments. First the agent is installed into the system, after security has cleared it, and then it should be started. Telescript [Gen95] has the agents continue with the instruction following the *go*-instruction. The go-instruction is Telescripts method for moving agents. AprilQ++ [CM95] has the target environment install the code and then call a *re-activate*-method to start the agent.

2.3.3 Security

There are several threats [Che] to an agent application. A simple way of categorise them can be:

- An agent threats the environment.
- An environment threats an agent.
- An agent threats an other agent.
- An environment threats an other environment.

One could argue that there only exists two categories, agent threats environment and environment threats agent, but this would put distinct issues into the same category.

Communication is also a problem. In open systems, measures must be taken so that messages between two nodes are not forged, changed, tapped, removed or in any other way tampered with.

Agents threatening environments

The first threat in agent applications is that an agent tries to modify the behaviour of the environment in some way. In a sample scenario the agent would simply make the environment fail (crash). It is easy to realize that the environment needs some kind of protection from this, especially since one of the wanted properties in many agent applications is availability of service. This threat is important and should be a key issue when designing of the whole agent environment.

What allows the agent to threat the environment can be reduced down to the agents description. If the description contains nothing which can cause damage, it is safe to execute the agent. If the description does contain something which might compromise the security of the environment, this must be handled. Suggested ways to handle this is:

- Designing the language so that all unsafe constructs are removed, or executed in a safe interpreter like SafeTcl [LO95].
- Verifying the incoming description. As an example Java verifies byte code before it is loaded.
- Run-time checks of the execution. Java also makes run-time checks that the description for example does not index outside arrays.

The agents must also be scheduled in a fair way so that they can not monopolise the CPU, and the amount of system resources they use must be controlled.

Environments threatening agents

The second threat is that the environment executing an agent reads or changes the description or knowledge of the agent. This threat is probably impossible to handle in a reasonably efficient way without using extra specialised hardware, Knabe [Kna95] discusses this[1]. More feasible methods would be to use authorisation or to have some kind of gossip telling which systems to avoid.

One related issue is that the operation of an environment can be disturbed in other ways than by agents. Failure of the nodes in a distributed network should be expected and ways of reporting that agents die, or are lost, in an abnormal ways are most likely to be needed. A scheme where *rear guards* [JvRS95] are left in the trail of an agent, i.e. when the agent leaves a base it leaves an agent behind which monitors the status of the new agent at the next base, can be used to manage this problem. The rear guards detect if the connection to the next or previous incarnation of the agent is lost and take the appropriate action. This kind of protection mechanism is however not the environments task but the applications.

[1] It seems fair to say that the usefulness of such systems would be severely limited.

Agents threatening agents

That an agent harms another agent is the third threat. The harm can be that the agent

- mounts some kind of denial of service attack; or
- modifies an other agent.

As examples of denial of service attacks, agents could kill the execution environment or kill the other agents. They could fill the mailboxes of others preventing other agents from communicating with them. Filling mailboxes can also make the agent use all of some resource of which it has been given limited amount, e.g. execution time. Denial of service attacks are hard to solve since they can be implemented using, for example, communication which the agent system requires.

The easiest way to modify another agent it to lie to it, and thereby try to change its beliefs and knowledge. The protection from this is application based and should not be a part of an agent environment. That an agent modifies the code or knowledge of another agent in other ways must however be prevented.

Environments threatening environments

The category where environments hurt each other is a common distributed systems problem. An environment could pose as an other environment to increase its access rights to a third environment. This kind of problems suggests that some kind of authentication is needed together with secure communication. There are several methods of solving this, but it will not be the subject of this paper.

Scope of this paper

Of the threats described above, this paper will present means to manage agent-agent and agent-environment threats. The environment threats environment problem will also be be dealt with, but not as fully or clearly. This protection will come from the fact that references to other environments are made into unforgeable capabilities with restricted access rights.

The safest way to protect agents from environments is to make them avoid visiting untrusted environments before they are convinced that the environments will not hurt them. To be perfectly safe an authentication mechanism is needed to verify the identity of the environment. This will be further elaborated in this paper.

Chapter 3

Erlang

This chapter gives a short introduction to the parts of the ERLANG[1] language and runtime environment which can be used, or will have to be modified for the programming of mobile agents.

ERLANG is a process oriented functional language designed for programming concurrent, real-time, distributed fault-tolerant systems. Some features supported or easily implemented are

- continuous operation;
- robustness;
- memory management; and
- distribution.

The language has been developed in the telecommunication world, a domain where many applications need these features. As an example it is well suited for the implementation of telecommunication switches.

3.1 The distributed runtime system

When an ERLANG distributed runtime system is started it becomes a distributed node. These nodes are the execution environments of ERLANG

[1] A full description of the language can be found in [AVW$^+$96] in union with the source code.

programs. A node is addressed by its globally unique name. Nodes can be connected in an all-to-all way to form a distributed system. In the nodes processes can be started to do computation.

The nodes have their own name space, for registering process names, and there is only one name space per node.

A process is accessed using its *Pid* (**P**rocess **Id**entifier). The Pids are globally unique and can be used in any part of a distributed system to address the process.

3.2 Communication

Communication has been a key issue when ERLANG was designed. It is implemented as asynchronous message passing. The recipients of messages are the processes. There are however two kinds of messages in the system,

- normal messages created using the message passing mechanism; and
- special runtime signals.

Each process has a mailbox and all messages sent to that process are stored in this mailbox in the order they arrive. The message passing guarantees that if the messages are delivered from one process to another, they are delivered in the same order as they where sent.

A process is addressed using its Pid or a name the process has been registered under. Processes can only be registered under one name. Registration is node local but it is possible to specify names on other nodes using a special name format.

The reception of messages is simple. A receive primitive gives a number of patterns and an action taken for each pattern. When the execution reaches the receive statement it tries to match the first message in the mailbox against the patterns in order. If the message matches a pattern the action associated with that pattern will be taken. If the pattern contains any unbound variables these will become bound with corresponding parts of the message. If the first message does not match any pattern, the second message in the mailbox will be tried. The first message will however not be removed or lose its position in the mailbox. If no message matches the patterns the process executing the receive will be suspended until a message that does match a pattern arrives. The receive primitive can be given timeouts with actions to be taken.

There also exists signals which are system information, mostly errors. The signals are delivered using a mechanism similar to the one for message passing but by default there exists no way of conditional reception. Some signals can however upon request be transformed into messages and they can then be treated like messages. Using this it is possible to monitor processes and get messages when they quit executing, whether it is normally or abnormally.

3.3 Transport

The transportation was in the previous chapter divided into two separate parts, the knowledge transmission and the agent description transmission.

As Erlang is a functional programming language it becomes natural to carry around a state as an argument to the process. This state can easily be viewed as the knowledge of an agent.

The description transmission is a bit trickier. It is today possible to execute code on other distributed nodes by sending code to another process. The code can be transmitted as byte code or as a lambda expression. The functionality of the remotely executed code is however limited to one module or expression. Something that is not supported is auto loading of remote modules, i.e. that when a module is executed on a node different from its node of origin, it triggers auto loading form the node of origin and not from the current node. Without support for this type of distribution of code, agent programming in Erlang would be cumbersome.

3.4 Security

There is only one security construct in the language dealing with distribution. It is a cookie protocol which is used in the communication between distributed nodes. Nodes must have the same cookie set if they are to be able to communicate with each other. This means that if three nodes are to communicate they will all have to have the same cookie. This scheme makes it tricky for a node to communicate with two other nodes which in turn are not allowed to communicate with each other. In our earlier categorisation this can be seen as a environment treating environment solution. Nodes can today only communicate if they have set the same cookie.

There is no security protecting the run-time system in any other way. Neither is there protection of the processes. This is however reasonable since the language has been used in closed systems where all components have been well specified.

Chapter 4

Design

This chapter first identifies the weaknesses of Erlang today. Then the design of extensions to remedy these weaknesses are described.

There are different ways in which security can be incorporated in a programming language. Either the language can be modified so that it contains no insecure constructs, as in Safe Tcl [Bor94], or the language can be run in a secure environment.

The approach used in this paper is to modify the language into a safe and secure language, not to create a safe sub or super set of it. The language is extended and parts of it are modified to better work with the extensions. Together the changes better control the security of the environment.

4.1 Identifying weaknesses

The parts in the Erlang environment that have to be made secure are easy to identify. The threats from processes, c.f. chapter 2, comes from that the processes can

- modify the run-time system; and
- modify the state of other processes.

The solution is to put restrictions on the right to do this.

As mentioned earlier, the other threats outlined will not be addressed. The environment environment threat will be handled in much the same way as the “agent threats environment” threat and we make some assumptions below that further reduces this threat.

Assumption 1 The communication channels between different run-time systems are secure. This might not be the case today, but with the current demand for this and with the current developments in the area, e.g. SSL, it is reasonable to assume that channels in a near future will be secure. This assumption leads to that the communication in this paper will not be encrypted by the run-time system.

Assumption 2 All communication passes through defined channels, i.e. there are no secret channels connecting to a system without it knowing about them. This assumption leads to that systems can scan and, if need be, modify all incoming and outgoing communication.

4.1.1 Security

From now on we consider the Erlang processes as possible agents and treat them as the same.

We start by describing how the different threats presented in the previous chapter appear in Erlang and then we give means of eliminating them.

Processes threatening the run-time system

There are two ways in which processes can change the behaviour of the Erlang run-time environment.

- The run-time environment can be halted using the built in function `halt()`.
- The state of the run-time environment can be changed by using resources like memory, reductions[1] and the number of processes running in the system.

Note that the resources in the second threat have something in common. They, in some aspect, belong to the runtime system, not the processes. A runtime system is, for example, limited to a given amount of memory and a maximum number of processes.

[1] The measure of computation in Erlang.

In the application ERLANG was designed for, these threats are not present since these applications often are closed systems where all code is trusted. There is no need for protection from processes executing the halting primitive, the only ones doing it are designed to have the functionality. In a system where all process may not execute this command a policy for the built in functions usage must be introduced.

That the run-time system can be harmed by processes using resources demands a bigger extension. The resources available can be divided into two categories:

- System resources like memory; and
- application resources like registered names.

To handle the problem of system resource consumption it must be made possible to put some kinds of restrictions on this usage. Such restrictions could be that an application is not allowed to use more than 10Mb of memory or that it may not have more than 100 process executing in parallel. This kind of restriction is actually present in todays ERLANG system. It has a limited amount of memory and a limited number of concurrently running processes available to it. If the run-time system tries to exceed these limits, it fails and is shut down.

Processes threatening other processes

It is easy for processes to harm other processes in the current implementations of ERLANG.

- Processes can be harmed using their identifier, Pid, and
- the code a process uses can be modified.

Processes are identified and referenced using their process identifier, Pid. There exists a built in function which returns a list containing the Pid of every process executing in the same node as the calling process. Pids are used for all purposes, e.g. killing processes, getting information about them and sending messages to them.

A more elaborate way for processes to harm other processes is if the code they use becomes modified. ERLANG supports run-time loading and swapping of code and this makes it easy to utilise this threat. What makes this possible is that all processes share the same name space and directly can swap or unload loaded code. This problem will be handled using a combination of subnodes and a new code loading mechanism.

Since ERLANG nodes have only one name space, processes can register names which other applications may require. This could lead to that an application never can register a name needed.

4.2 Capabilities

The first extension aims to make the identifiers used to access objects more secure. The discussion will be about processes but the extension can and is be used on identifiers to other objects as well, e.g. node identifiers.

The processes are referenced using *process identifier*, Pids. In todays ERLANG there are no restrictions on how these identifiers can be obtained or used. If not found in another way, the Pids can be created from strings. In an open agent environment this is not acceptable. Both the ways of obtaining the identifiers and the way in which they can be used must be restricted.

The problem is very similar to the one operating systems have with issues like access to resources and memory addressing and protection. Capabilities [Rob83, Mil92] have sometimes been used to provide operating systems with a single, unified mechanism to handle these issues. This mechanism can be used for the protection needed in ERLANG.

4.2.1 Representation of capabilities

Capabilities have two parts, the identifier and access rights. The access rights specifies how the capability can be used to access the specified object.

Identifiers

The identifier in a capability references the object which the capability can be used to affect. The current Pids can be used as this part of the capability.

Access rights

The choice in this paper has been to have the identifiers carry the information about their allowed usage with them. This removes the need for nodes to remember all granted capabilities concerning objects on the node.

In a file system the objects identified by the capabilities are files and the access rights needed are rights like read, access and write. In object oriented languages identifiers are the methods of the object and the rights are ones like public and private. When processes are referenced the choice of access rights becomes more complicated. If you hold an identifier to an objects communication mechanism, you should be able to communicate with that object. This might suggest capabilities with only one access right each. This is not a reasonable solution in Erlang since it would be awkward if processes had to remember more than one reference to other processes. If they had to, the extension would gravely change the language.

The solution described in this paper has the access rights represent each of the built in functions applicable on the identifier and place them in a list inside the capability together with the process identifier. Processes have their set of built in functions affecting them and they have these functions as access rights. If file descriptors are to be made safe, their identifiers are turned into capabilities and the access rights in these set to represent the functions operating on files. Nodes and other objects accessed through capabilities also can get their own tailored set of access rights.

As an example we see that capabilities can be used to restrict the usage of the halt primitive simply by demanding that the node affected is identified.

Encryption

The run-time system can be modified to guarantee that a capability may not be locally altered, i.e. it can prohibit the locally executing processes from changing the access rights of capabilities. When the capabilities leave the safety of the node where they where created, i.e. the only trusted node, they need protection so that they are not altered by other processes or nodes.

We have chosen to encrypt the capability. The parts containing the identifier and the access right are encrypted together so that they can not be forged, partly replaced or substituted. This does however impose some awkward restrictions on the usage of capabilities. Simple comparison can not be used to decide whether two capabilities identify the same object.

4.3 Nodes

The second extension makes the run-time system both more secure and also introduces new was to make applications run safer.

If agents could crash the node they execute in or use up a resource, they effectively could deny other agents the services provided by that node. ERLANG has today no protection at all against these kinds of attacks.

The resource issues have been solved in Telescript [Gen95] which has a scheme where permits are used to manage resources. This solution is based on that there exists a base for paying for service and since this base is not present, perhaps even should not be present, in ERLANG, an other mechanism must be found.

When the run-time system is started today it becomes a node. This node has a set of resources bound to it in an indirect way. It has for example a limited amount of memory available to it. Should it try to use more memory the node will fail and shut down. The failure is detected by the runtime system but there is no way to stop the system from shutting down.

This concept of nodes will just need to be slightly modified to give a more general mechanism for security and management of resource usage. The solution presented here is a scheme where an arbitrary hierarchy of nodes can be built. Constructs for creating child nodes and managing nodes are added to the language. Child nodes share their parents resources. If a node violates the usage of a resource it is shut down, and any children nodes of nodes shutting down are also shut down. The runtime systems become top nodes since they do not have any parent node.

Nodes are started with restrictions on their resource usage. How the restrictions should be applied is discussed below.

Resources

The resources available in nodes are system resources, e.g. memory or disk space, and processes. It is possible that system resources are shared between several top nodes but the processes can never be shared between ERLANG nodes, i.e. between ERLANG run-time systems.

If a node has its usage of a resource restricted and it is detected that this restriction has been violated, the node is shut down. To supply the means for nodes to guard their resource usage, and monitor usage so that shut down can be avoided, functionality has been added to the `node` module which returns the resource status of a node.

The restrictions do not however ensure the availability of a resource. To ensure that a subnode always has, for example, 10Mb of memory available to it, would also ensure that no other node could use that memory. This is not an effective way to use memory and should not be encouraged, hence the scheme does not allow this to be described[2]

Name spaces

Names, i.e. registered processes, and modules are node local. It is possible for other nodes to address registered names, but they can not register names or load modules inside a node from the outside.

With child nodes the issue of scope and visibility arises. Should child nodes see names registered in parents or vice versa? The answer is that neither should be the case. This can easily be illustrated with a small scenario.

A node creates a subnode and starts an application inside it. One of the nodes, say the child node, registers the name security_server. Now, when the child node has done that, the parent node registers the same name. The child node now unregisters the security_server, but alas, the name is still available to processes in the subnode. Now imagine what can happen if the functionality of the two servers differ. In the other direction, where parents see the names registered in child nodes, the problem of name spaces is not solved in the parent nodes, they will still have to deal with all names in side them and they will be forced to run conflicting applications in separate subnodes.

Security

The first threat identified in chapter 2 was the threat from processes. This threat came from that processes could halt the system or misuse its resources. The subnodes prevent the resources from being violated and the capability mechanism can be used to protect the nodes from being misused in other ways. The use of capabilities as identifiers of the nodes will make the access to them manageable.

[2] The property can however only be described if the top node has a guaranteed amount of the resource available to it.

The second threat comes from that other nodes or processes on other nodes try to misuse resources in a node. The only way for other nodes to reach the node is through communication channels so the nodes will need to have their communication managed in some way. The capabilities are to be encrypted outside the node so there should be a standard way to communicate with the node, and the processes inside it.

4.4 Distributed code

Todays code loading mechanism in ERLANG is not suited for mobile agents. The two main reasons are that

- auto loaded code is only searched for locally and
- the loader is not secure so malignant code can be loaded into the run-time environment.

The requirement that code has to be available on local disk to be auto-loaded can easily be changed. By modifying the code loader, code can be loaded from an arbitrary place to the cost of added need for security. Code loaded from another system must, as earlier stressed, be checked so that it will not threat the security and safety of the own system.

4.4.1 Managing code with Mids

If code is to be moved around in a distributed system the need for some kind of package to hold the code becomes evident. The structure introduced here is the *Mid* (**M**odule **id**entifier). A Mid contains information on the origin of the code and some kind of representation of it. The representation can be everything from byte code or source code to a reference. There are different properties of the code format.

- If the code is sent as byte code it will be fast to load when need be but will demand that the loading mechanism is made safe so that the run-time system can run the code without the need of run-time security checking.
- Using source code, or a parsed representation, will make security easy since the code easily can be checked and compiled using a trusted com-

piler. The fact that the code will have to be compiled will obviously lead to that this format if slower than using byte code.

In SafeErlang the parse tree representing the code is used as the transfer format. This enables each node to compile any code using a trusted compiler, thereby assuring itself that the code is safe.

There are also other gains when using Mids. It becomes possible to have several versions of the same code running in the same node. In Erlangs current implementation only two versions of a module can be loaded in the system at a given time, the old and the current version. When a newer version is loaded the current version will become the old version and the old will be unloaded. This would be awkward in an agent application since there might be agents who live longer than two changes of a module. These agents would be destroyed should the current implementation of the code loading be used.

Mids in a distributed system

There is however some more aspects of the code mechanism that will have to be modified to make the use of Mids in a distributed Erlang system comfortable.

When a Mid is used on a node other than its origin and a call is made to a remote function from inside it the module referred is probably a module located at the current modules node of origin, and not the module located in the current node. If for example an agent moves to a node and calls a function in the module `maintenance` it is most likely expecting the module from its origin node and not some module available in its current environment. The module form the node of origin will therefore have to be loaded into the node where the agent executes.

If this is the case other problems rise. How are modules from other nodes used? If a process is started at a node using code from another node, how can it access node local code? There is one obvious way which makes the agent communicate with a local process and having that process sending a Mid with the code. Another way is to let the nodes register modules and then making registered modules available to processes executing in the node.

Registered modules

If a module is registered, that module will take precedence over other modules with the same name in that node. Call to modules of the same name as a registered module will always be treated as calls to the registered module. This is done regardless of if the process calling the module has another module in mind.

This mechanism makes it possible for nodes to supply a set of specialised module instances to executing processes. As an example a node can register its own version of the module for file management. Processes will then call and use this module instead of the standard file module.

Safe standard modules should also be registered or else processes executing in the node would bring, and install, their own instances of the standard modules. This would lead to that the node could have several loaded versions of a given module.

Chapter 5

The Prototype

The prototype implements a fully functional runtime environment with capabilities, a hierarchy of nodes and a modified code loading mechanism.

The changes are in the compiler and not in the emulator. The emulator is ever changing and the prototype would not be able to take advantage of future improvements if parts of it were implemented in the emulator. It would however be possible to make it more efficient than the current compiler based implementation should it be in the emulator.

The changes in the compiler are made in its pre-expansion pass. This pass is used to implement transformations of the parse tree enabling, for example, records and lambda expressions. The pass can easily be extended to handle other constructs as well. The changes made redirects all calls to built in functions to function calls in the module `kernel`, making it possible to modify the behaviour of the built ins. Some test predicates are changed to handle new arguments, e.g. the Pids have changed into capabilities so the test `pid` must be trapped and its behaviour altered so it can test whether or not the capability supplied is a process identifier.

5.1 Capabilities

5.1.1 Using capabilities

Capabilities are used to reference objects in the environment. Mainly the usage of capabilities is identical to the usage of current Pids. The main difference is that when a capability is used in a way not granted by its access rights, the process over using it will be notified (by an exit signal).

Creating and obtaining capabilities

There are three ways of obtaining capabilities.

- Executing the built in function `self()`, which returns a reference to the calling process, will return a full capability containing all available access rights,
- creating a new process will also return a full capability (to the new process) and
- a capability can be created with arbitrary access rights by a process which has the required access rights to the node the capability belongs to.

Other built in functions returning Pids like `whereis`, which given a name returns the associated process' identifier, and `processes`, which returns all processes executing in the same node as the caller, will return empty capabilities, i.e. capabilities with no access rights. The reason for not removing these functions is that system processes should be able to, for example, collect all processes in a node and then manipulate them. In order to do this the system process would however need to add rights to the empty capabilities first. A more detailed description of how this could be done can be found in appendix B.

Capabilities can also be obtained by message passing or by shared memory. This is not considered a security problem since the process sending or storing the capability, obviously, only can do so with a capability it already possesses. The usage of obtained capabilities is up to the application, not to the Erlang system.

Viewing capabilities

To view the part of a capability a new built in function, `view` is been introduced. A call with a capability given as argument returns the identifier of the process referenced (i.e. the Pid), the node the process executes on and list of access rights (i.e. names of the built in function with which the capability can be used).

Restricting capabilities

To restrict the access rights of a capability the primitive `restrict` with two arguments is introduced. The first argument is a source capability and the second is a list of access rights. If the list of new rights is a subset of those present in the capability, a new capability will be returned containing only the subset of rights. If the list is not a subset of rights the function will fail.

Executing the built in function will not modify the source capability.

Comparing capabilities

Security makes it necessary to have the capabilities encrypted. This does however make it difficult to compare, and match, two capabilities to see if they refer to the same process. Both capabilities will have to be decoded first, and the only ones who can do this are the nodes the capabilities came from. If they are local, i.e. they have not been sent outside the node, this can be made fairly cheap but if both capabilities from other nodes it becomes expensive since communication will be necessary.

The functionality can not be implemented as a test. Tests are used for clause selection and have restrictions on what they can do. They may for example not have any side effects and since there might be communication involved in resolving the different part of a capability, the function can not be implemented as a test. The solution has been to implement it as a built in function which shorthands the two `view` calls needed. The comparison function is called `same` and it returns true if both capabilities given as arguments identify the same process, otherwise false.

This problem could be eliminated with a space sacrifice. If the capabilities where extended with unencrypted information, for example the process identifier and the access rights, the comparison would be reduced to a simple equality check.

5.1.2 Revocation

One facility desired in a capability based system is the availability of capability revocation. How this can be implemented in a system where no database containing valid capabilities might not be obvious. When a capability is revocated it becomes invalid and further usage of it will not affect the identified process. This can be implemented using the capability scheme described here simply by giving the responsibility of revocation to the application. If the application wants to implement revocation it can start extra processes which just works as duplicates of the original process. These duplicates can then be destroyed to revoke any capabilities granted through the use of that process.

Another way to manage revocation would be to use names, circumventing capabilities altogether. Revocation would then be unregistering the name.

5.2 Nodes

5.2.1 Using nodes

There is currently one way to create a node, to start an Erlang run-time system. This creates a top node. To create sub nodes the new built in function `subnode` is introduced. The function takes two arguments of which the first argument is the name of the new node and the second argument are options for the node. These options are the resource restrictions placed on the node[1]. The return value from the function is an identifier of the new node, a capability containing all access rights used with nodes.

The capability is used to access the node. This is the difference from the current Erlang implementations where the name of the node is used. Using the new scheme a node's name is useful for identifying the node, in for example node-down messages or other similar situations where the need for access rights is not present, only.

5.3 Distributed code

5.3.1 Using Mids

To read code into Mids the built in `read_module` is introduced. It takes a module name as argument and returns a Mid containing the code. The Mid can then be used wherever a module name can be used.

Modules called inside the Mid will be regarded as calls to the module at the Mids node of origin. If the called module name is registered in the node

[1] It is possible to have the options containing other information, such as the names of servers, as well. This is regarded as an extension of its own, outside the scope of this paper.

where call is done, the registered module will be used instead. Registration, and unregistration, of modules is managed using functions in the module `node`, see appendix B.

5.4 Implementation of the prototype

This section describes the experimental prototype that has been implemented. Most of the prototypes components are not needed if the system is integrated into the Erlang run-time system.

The prototype is divided into five parts:

- The kernel
- Nodes
- Gates
- The name server
- The code management

The built in functions are divided into two categories, safe and unsafe.

- The safe functions can be executed without any risk that the environment will be compromised. The main resource used in safe calls is CPU (although a little memory also can be used).
- The unsafe functions use one or more resources other than CPU. If the function is executed directly it might compromise parts of the environment so the call must be checked before the call is actually carried out. As an example we have to check if a process uses a valid capability when it tries to send a message. If the capability is valid we send the message, otherwise we report failure to the process.

The requests created in the kernel are sent to the node on which the calling process executes. How the nodes work will be described below.

Nodes

The nodes are simulated using processes. They maintain information about the resources currently used by them and the resources it has used. When a process is created inside a node the maintaining process of that node will register the process and keep track of the resources it uses. When a process wants to execute an unsafe function, the kernel turns the call into a request which is sent to the maintainer of the process' node. In some cases the request does not affect the node the process executes on and it can in some of those cases be sent to the maintainer of the node where the unsafe function tries to use a resource.

The resources used are checked by statistics gathering. The processes report resource usage when they die and if the node needs to check its current resource usage it can add this to the resources used by the currently executing processes. The resources used by the currently executing processes is collected using ERLANG s built in function for viewing the information about a given process.

Gates

To handle the communication between the nodes the notion of gates is introduced. The only way to send a message to a node (manager) is to send it through the gate of the node. The gate is the part of the prototype which checks that capabilities can be used with specified resources.

As functions can be used to operate on a remote node and the result will have to pass through the gate of the calling process node, an extra access right is needed. In the prototype this right is called reply and it can only be used to return a value to a specified process.

The code management

A code server is started for each top node. This server will load code into the run-time environment and it is used by the top node and all its children. Using just one loader solves the problem that several copies of the same code are installed into the system. To allow all nodes in the same run-time system to share code, code is installed under a unique name and calls are translated into these names before a call to a module is made.

It is however not required that all nodes inside and including a top node use the same loader. As with all servers it is up to the node to choose which server it actually uses.

Chapter 6

Conclusions and further work

6.1 Conclusions

The proposed expansions are a good complement to the language. Together the ERLANG language and the extensions form a strong environment in which safety critical and secure applications can be implemented.

Capabilities

If the size of the capabilities is limited so that it can not contain an unencrypted representation of its access rights comparison will become a problem. An integrated implementation would not need to have the capabilities encrypted while they are in the node where they where created, they would only have to be encrypted when outside that node. The unencrypted form would not be available to the processes, they still believe them to be encrypted, but the run-time system can effectively execute node local comparison and access right verification. Comparison on remote nodes would however require communication with the node of origin. If, on the other hand, size can be sacrificed for efficiency it would be possible to implement an efficient comparison of capabilities on any node.

The capability concept seems like a promising way of restricting the access rights to objects in the ERLANG environment.

Nodes

Nodes introduced the means of restricting the resource usage in a node. The concept is very close to the notion of nodes in current versions of the language.

The nodes also introduced name spaces, something missing today. Several problems in larger ERLANG applications come from the fact that there are name conflicts. This problem can be eliminated using subnodes.

Safety critical application can be run inside subnodes to ensure that they do not affect each others computation.

The new code loading

The new code loading enables code to be loaded from other nodes and not only from the local node. The code is sent as a parse tree representation and fully compiled in the node where it is used. This places all the responsibility in our own compiler. This seems like a feasible way to solve the problem with distribution of code.

6.2 Further work

There are several problems which are outside the scope of this paper.

Capabilities

This paper have presented a solution using encrypted capabilities. It has not presented how they are encrypted. There are several different cryptographic techniques that could be used. If parts of the capability contains unencrypted information, e.g. to enable more efficient comparison, the choice becomes even more delicate. Which technique to use should be studied.

Protecting more resources

The identifiers of processes and nodes have been turned into capabilities, but there exists other objects which should be capabilities as well. Files, ports and databases are a few examples. All these different object types must be identified and a scheme to separate their access rights derived.

Nodes

This paper has not described a scheme for authentication of nodes. The need for the cookie protocol used in the current versions of ERLANG has been eliminated by the proposed extensions but the protocol was not really used to solve the authentication problem.

Secure channels

Large parts of the design is based on one of the assumptions made in chapter 4, that the channels are secure. This makes it obvious that secure channels must be added to the ERLANG environment.

Code

The new code loading was implemented without modifications to the run-time system. This is however an awkward way of solving this problem. Small modifications need to be done in the run-time system to allow an arbitrary code loading mechanism to be implemented.

Code garbage collection

With the introduction of Mids one problem has been left aside, the need for garbage collecting old modules. In a distributed system where the source node always is available all modules currently not in use, i.e. not having a process executing them, can be unloaded. The only loss is the time when reloading modules. When the connection to the source node of a module becomes unsafe, i.e. it might not always be possible to reload the module, it however becomes a problem. If the code is removed from a system while it is still needed, processes executing it will not be able to complete their tasks and the module should not have been unloaded.

Bibliography

[AVW$^+$96] Joe Armstrong, Robert Virding, Claes Wikström and Mike Williams. Concurrent Programming in Erlang, Prentice Hall, 1996.

[Bor92] Nathaniel S. Borenstein. Computational mail as network infrastructure for computer-supported cooperative work. In *Proceedings of CSCW 92 (Computer-Supported Cooperative Work)*, 1992.

[Bor94] Nathaniel S. Borenstein. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *Proceedings of the 1994 IFIP WG6.5 Conference on Upper Layer Protocols, Architectures, and applications*, May 1994.

[CGH$^+$95] David Chess, Benjamin Grosof, Colin Harrison, David Levine, Collin Parris and Gene Tsudik. Itinerant agents for mobile computing. Technical report RC 20010, IBM T. J. Watson Research Center, March 1995. Revised October 17, 1995.

[Che] David Chess, Things that go bumpp in the Net, IBM Corporation, Available as http://www.research.ibm.com/xw-D953-bump.

[CM95] K. L. Clark and F. G. McCabe. April - agent process interaction language. In M. Wooldridge N. Jennings, editor, *Intelligent Agents, LNAI, vol 890.* springer LNAI, 1995.

[FFM$^+$93] Tim Finin, Rich Fritzson, Don McKay, and Robin McEntire. KQML: an information and knowledge exchange protocol. In *International Conference on Building and Sharing of Very Large-Scale Knowledge Bases*, December 1993.

[Gen95] Telescript Language Reference, General Magic, Inc. October 1995.

[GF92] Michael P. Genesereth, Richard E. Fikes, et al. Knowledge interchange format version 3.0 reference manual. Technical report, Computer Science Department Stanford University, 1992.

[GK94] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 147, 1994.

[Gra95] Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceeding of the CIKM Worshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, December 1995.

[HCK95] Colin G. Harrison, David M. Chess and Aaron Kershenbaum. Mobile agents: Are they a good idea? Technical report, IBM T. J. Watson Research Center, March 1995.

[JvRS95] Dag Johansen, Robbert van Renesse and Fred B. Schneider. Opertaing system support for mobile agents. In *Proceesings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, 1995.

[Kna95] Frederick Colville Knabe. Language Support for Mobile Agents. Tehcnical report ECRC-95-36, European Computer-Industry Research Centre, December 1995.

[LD95] Anselm Lingnau and Oswald Drobnik. An infrastructure for mobile agents: Requirements and architecture. Fachbereich Informatik (Telematik), Johann Wolfgang Goethe-Universität, August 1995.

[LO95] Jacob Y. Levy and John K. Ousterhout. Saft Tcl toolkit for electronic meeting places. In *Proceeding of the First USENIX Workshop on Electronic Commerce*, pages 133–135, July 1995.

[Mil92] Milan Milenkoviç. Operating systems concepts and design, second edition, pages 348–352, McGraw-Hill, 1992.

[Rob83] Dorothy E. Robling Denning. Cryptography and data security, pages 216–228. Addison-Wesley publishing company, 1983.

[Sun94] The Java language: A white paper. Sun Microsystems White Paper, Sun Microsystems, 1994.

[Sun95] Java Mobile Code, A white paper. Sun Microsystems White Paper, Sun Microsystems, 1995.

[TV96] Joseph Tardo and Luis Valente. Mobile agent security and Telescript. In *Proceedings of the 14th International Conference of the IEEE Computer Society (CompCon 96)*, February 1996.

[Whi94] James E. White. Telescript Technology: The foundation for the electronic marketplace. General Magic White Paper, 1994.

[Whi95] James E. White. *Mobile agents.* General Magic, 1995.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent Agents: Theory and Practice, January 1995.

[Yel96] Frank Yellin. Low level security in Java. *WWW4 Conference*, December 1995. Available as
http://www.sun.com/sfaq/verifier.html.

Appendix A

Unsafe built in functions

This appendix lists the unsafe built in functions, BIFs, of SafeErlang. New functions are introduced, some are modified to work in the new environment and some are rendered obsolete and hence removed.

The new BIFs

To support capabilities, the hierarchy of nodes and the new code loading, there is need for an extension of the built in functions.

New process and capability BIFs

`kill(Capa)` Is the new BIF to terminate the execution of processes. The old BIF used to do this was `exit(Pid,Reason)`, `Pid` now a `Capa`, with `Reason` set to kill but having `exit(Pid,Reason)` overloaded in this way makes it impossible to have distinct access rights for the two different behaviours of the function. The solution was to introduce the new built in.

`same(A,B)` Returns `true` if the arguments refer to the same process, else `false`.

`view(Capa)` Returns the tuple `{Id,Node,List}`. `Id` identifies the process, `Node` is the name of the node on which the process executes and `List` is a list of the access rights of `Capa` `restrict(Capa,List)` Returns a modified version of `Capa` having only the access rights listed in `List`. The new access rights must be a subset of the access rights in `Capa`.

`Id`, returned from `view`, is the identifier of a process. This identifier is only good for comparison and can never be used as anything else.

The lists of access rights contain the names of the built in functions the capability can be used with.

New distribution BIFs, node BIFs

`subnode(Name [,Options])` Creates a subnode and returns a reference to it. `Options` is used to restrict the resources used by the node.

The resource usage is restricted by the options. These restrictions do not guarrantee availability. However it is guarranteed that the node will be shut down if the restrictions are compromised.

The options given to nodes are[1]

- `{average_load,Limit}` where `Limit` is a float indicating the maximum CPU load the node will exist under. If the load exceeds the limit, the node will shut down.
- `{memory,Limit}` where `Limit` is an integer indicating the maximum memory usage of the processes inside the node.
- `{processes,Limit}` where `Limit` is an integer indicating the maximum number of processes running in parallel inside the node.
- `{reductions,Limit}` where `Limit` is an integer indicating the maximum number of reductions the node may use.

New module BIFs

`read_module(Module)` Returns a Mid containing the current disk version of the module `Module`. This Mid will be different from other instances

[1] The options are implementation dependent, this is the set implemented in the prototype.

of the same module if the code is not identical, i.e. if the source has changed a new Mid will be generated.

`modules()` Returns a list containing all modules visible to the calling process. Visible modules are the ones it uses privately and those registered in the node.

Registration of modules is described in appendix B.

The modified BIFs

There are several built in functions that need to be modified to handle capabilities or Mids. These functions are described in this section.

`apply(M,F,A)` Extended so it can use Mids.

`spawn(M,F,A)` Extended so it can use Mids.

`spawn_link(M,F,A)` Extended so it can use Mids.

`spawn(N,M,F,A)` Extended so it can use Mids and now uses a capability as its first argument.

`spawn_link(N,M,F,A)` Extended so it can use Mids and now uses a capability as its first argument.

`exit(Capa,kill)` This call is turned into a call to `kill(Capa)` so that it can get an access right distinct from other calls to the function.

`whereis(Name)` Returns a capability without any access rights.

`list_to_pid(List)` Like `whereis(Name)`.

`processes()` Like `whereis(Name)`.

`unregister(Capa)` Replaces `unregister(Name)`.

Note that `whereis`, `list_to_pid` and `processes` are kept and not remove only because that they are convenient while administrating a node. It then is handy to have identifiers to all processes running in a node available.

The removed BIFs

There are several built in functions that are no longer needed when the new extensions are used. Most of these handle the modules or the nodes.

Removed module BIFs

The following functions are not needed when Mids are used (their functionality has to be handled using other methods, e.g. garbage collection):

`delete_module(Module)`

`load_module(Module,Binary)`

`module_loaded(Module)`

`purge_module(Module)`

Then there is one function which lacks a meaningful result when Mids are used, so it is removed too:

`pre_loaded()`

Removed distribution BIFs, node BIFs

With the hierarchy of nodes and the assumption that there are no channels the node does not have control over, c.f. chapter 4, several BIFs handling nodes can be removed.

`alive(Name,Port)` Not needed when all nodes are supposed to be distributed.

`halt()` Replaced by shutdown in module **node**, see appendix B.

`disconnect_node(Node)` Like `alive(Name,Port)`.

`get_cookie()` Handled by the gates.

`set_cookie(Node,Cookie)` Handled by the gates.

Removed process BIFs

The built in function with process functionality are removed since they, potentially, only affect other processes, not the caller.

```
group_leader(Leader,Pid)
```

```
check_process_code(Pid,M)
```

Appendix B

The node module

This module is for accessing and altering the nodes at runtime. Most functions require that a Pid with the correct access right, N, is given as an argument, otherwise the process executing the call will exit due to a violation.

New BIFs in the node module

`shutdown(N)` Shuts the node down.

`create_pid(N,Type,Id,Capa)` Returns a capability referring to `Id` with the access rights of `Type` present in `Capa`. As an example, suppose that we have a capability to a node `N` and that the capability contains the access right create_pid in its list of access rights. It would then be possible to use `N` to modify capabilities from node `N`. The process capability `P`, belonging to the node, could be changed using the call `create_pid(N,pid,P,[send,kill])`, creating a capability with the access rights send and kill.

`register_module(N,Name,Mid)` Registers the Mid in the node.

`unregister_module(N,Name)` Unregisters the name.