A Platform for Secure Mobile Agents

Johan Arthursson[3], Jakob Engblom[3], Ing-Marie Jonsson[1], Rehan Mirza[3],
Gustaf Naeser[2], Mikael Olsson[3], Robert Ottenhag[3], Dan Sahlin[2],
Maria Schmid[3], Bertil Spolander[3], Elham Zolfonoon[3]

Medialab / Computer Science Lab.
Ericsson Telecom
SE-126 25 Stockholm,
SWEDEN

steam@cslab.ericsson.se

April 18, 1997

Abstract

A secure system for mobile agents written in ERLANG, a real time high-level functional language with processes and message passing, is described. The emphasis for this system is on security with reasonably secure nodes and secure agent communication, something which is often ignored in most agent systems, but crucial for an open agent system.

We argue that ERLANG is suited for programming mobile agents, because it already has the main features needed: processes, communication, and distribution. However, some extensions to ERLANG are needed, in particular in coping with the security aspects, which are discussed in the paper.

To validate our approach, a meeting-scheduler using mobile agents was implemented on our platform.

Most of the work reported was carried out as a student project at the Department of Computer Systems at Uppsala University. The name of project was *STEAM*[4].

1 Introduction

There are a lot of systems that offer support for mobile agents. AgentTCL[8] is an extension of TCL from Dartmouth College; Telescript[16] from General Magic; and Facile[11] where an existing language is modified to support mobile agents.

Most of these systems are either implemented as new languages or implemented in languages where functionality for concurrency, communication, fault-tolerance etc. has been added. We started with ERLANG[1], a language that already offers built-in functionality and mechanisms well suited for agent applications: processes, asynchronous communication, distribution, soft real time support and fault-tolerance. The emphasis on the project was on how to use the built-in facilities in ERLANG to acquire the required security, and to investigate how ERLANG needs to be altered to preserve security.

[1]Ericsson Medialab

[2]Ericsson Computer Science Laboratory

[3]Member of the STEAM project team at Uppsala University, Sweden. The other members were: Saleh Ali, Fredrik Eriksson, Fredrik Gihl, Erik Klintskog, Mohammad Mohammadi, Hishmat Mustafa-Sultani, Peter Nyström, Daniel Palmgren, Anna Sandberg, and Henrik Swerin.

[4]**S**ecure and **T**rustworthy **E**nvironment for **A**gent **M**obility

This paper is structured as follows: after a brief system architecture overview, our secure platform for mobile agents is described, followed by a brief description of our application, a meeting scheduler. The paper ends with conclusions and related work.

1.1 System architecture overview

Our system, which is named *STEAM*, consists of stations and agents where the agents are divided into mobile and stationary agents. Every station has a stationary agent called *station master*. The station master is responsible for the security on the station and it is the station master that controls all agents on the station, for instance when the agents enter or leave the station or try to communicate.

Communication between all agents uses KQML[4]. An agent may however choose to use an application specific protocol for other communication, for example when communicating with a user.

To be able to monitor the system, each station master has its own web page which is visualized using Java applets[7]. Through the web page you can easily monitor the agents on the platform and get information about their owners and what the agents are doing.

Java applets are also used for the user interaction with our application which is a meeting scheduler.

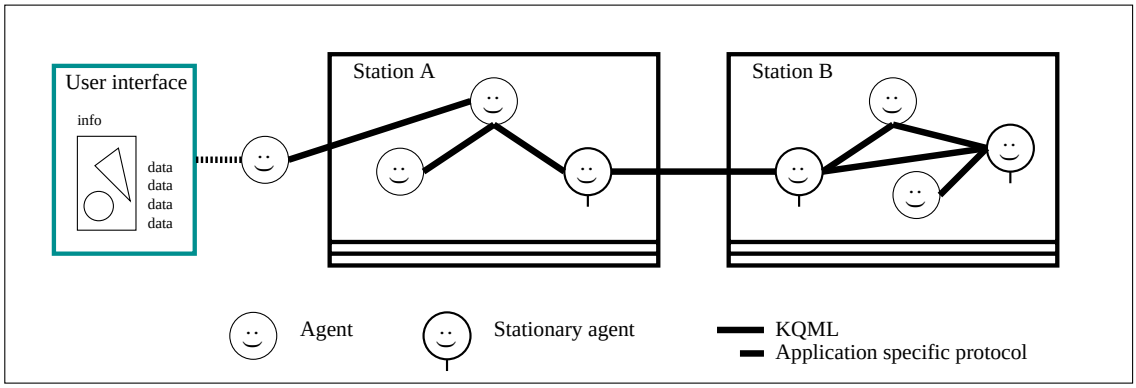

Figure 1: Agent system architecture

Each agent is divided into four parts: Agent Supervisor, KQML Gate, Agent Dictionary, and Agent processes. The first three are provided by the system whereas the last one is programmed by the user.

The *Agent Supervisor* is the part of the agent that controls its behavior, and mainly supervises the resources of the agent and charges the agent for its activities. The *KQML Gate* is responsible for making it possible for the agent to communicate with other agents. The *agent dictionary* serves as a shared memory for the agent.

2 A secure platform for mobile agents

2.1 Requirements

The main requirements for secure mobile agents relevant to our application are:

Requirement 1:	Stations must be able to protect themselves from potentially dangerous and destructive agents.
Requirement 2:	The system should restrict the resource usage of an agent to prevent it from monopolizing a station or from spawning an indefinite number of copies of itself.
Requirement 3:	Agents must be able to exchange sensitive and confidential information without the information being changed or monitored by a third party.

Additionally it is desirable for an agent to be able to trust that a station does not alter the behavior or data of a visiting agent. This however seems virtually impossible to implement just using software.

The STEAM system addresses the above requirements in different ways. In short:

- **Requirement 1**
  The agents are compiled differently if they are stationary or mobile, where the mobile agents are denied access to potentially dangerous modules or system features.
- **Requirement 2**
  A supervisor is attached to every agent that controls its resource and service usage.
- **Requirement 3**
  Encryption and authentication is used for agent communication.

In the rest of this section we will describe the implementation in greater detail. But first a few words on ERLANG, our implementation language.

2.2 Erlang

ERLANG is a fairly clean functional language: it does not have global variables and no destructive assignment. State changes are modeled by processes.

In ERLANG, processes are created explicitly through the `spawn` primitive. The result of the `spawn` is a process identifier. The process identifier is a reference to the created process and is used to address it, e.g. when sending messages to the process. On a Unix system all ERLANG processes typically reside in a single Unix process, and the ERLANG run time system schedules all processes in a fair way.

The ERLANG system supports distribution since version 4.0, i.e. processes may be spawned on remote nodes. For a large class of applications, it is quite transparent on which node a process resides, except for performance, as the send operation works equally well for processes on distributed nodes. The main difference is that it is possible to specify on which node a spawn operation should be performed.

A certain degree of security is provided through a *cookie* mechanism. All distributed ER-

LANG nodes that need to communicate must know each others cookies. The cookies work essentially like passwords. It should be noted that this protocol has serious weaknesses.

Once started on a certain node there is no direct way for an ERLANG process to move. If a process wants to move, it may spawn off a new process and then terminate on the local node. The state is carried over through the arguments to the spawned process.

2.3 Meeting the requirements

In the distributed ERLANG system it is quite possible for a process to misbehave in various ways. It can for instance find the list of all ERLANG processes and kill them. It can spawn off thousands of processes stealing all processing power, or it can allocate large amounts of memory etc. In the following sections we will see how these problems are solved in STEAM.

Agent closure

Our solution to the above mentioned problems is divided in two parts. The first is to encapsulate the agent in an agent closure. An agent can consist of several ERLANG processes and these processes execute within the agent closure.

The closure consists of three main parts. First there is an Agent Supervisor which is responsible for the agent, serving agent processes and handling charging for resource usage. At regular intervals the Agent Supervisor checks how much resources the agent processes are using. Typical resources are the CPU usage, i.e. in the number of reductions of the virtual machine in the ERLANG system and the amount of memory which is used by each agent process. When the agent cannot pay for its resource usage the agent will be terminated. With this approach an agent can only use up a limited amount of resources, if it wants to survive long enough to fulfill its goals.

To prevent an agent from being equipped with too much resource use credit, the payment for resource use has to be connected to some sort of real money, e.g. electronic cash.

The second part of the agent is the agent dictionary which acts as a shared persistent memory for the agent processes. The agent dictionary follows the agent when it travels between stations.

The last part of the agent closure is the KQML Gate through which the agent can communicate with other agents.

Agent compilation

It must be possible to prevent an agent from using functions that kill other processes, access the local file system etc. Still, some agents must be able to access local system resources in order to enable the agent system to perform meaningful tasks. To this end, agents are divided into two classes: stationary and mobile.

Stationary agents can't move and are started by the user on his own station. They are considered trusted and should not have any restrictions in using modules or BIFs.

Mobile agents, on the other hand, represent a potential threat and should not have unrestricted access to dangerous BIFs and modules.

To accomplish this the two classes of agents are compiled in different ways. Mobile agents are compiled with added runtime checks for module and BIFs usage. If a mobile agent tries to use a forbidden BIF or module, it is terminated.

Agent representation

An agent is represented as a package in which information concerning the agent is stored, e.g. agent type, unique agent id, agent dictionary etc. The agent code is also stored within this package in a precompiled form. This is essentially a parse tree created from the agent ERLANG source code. When an agent arrives at a station, the station master compiles the code and loads it into the ERLANG runtime system.

2.4 A library of agent functions

We have chosen to implemented agents in ERLANG as groups of communicating ERLANG processes. These processes can then perform specific tasks, such as communicating with other agents or performing computations. This is the natural way to design applications in ERLANG, and the language influenced the basic architecture of the agents.

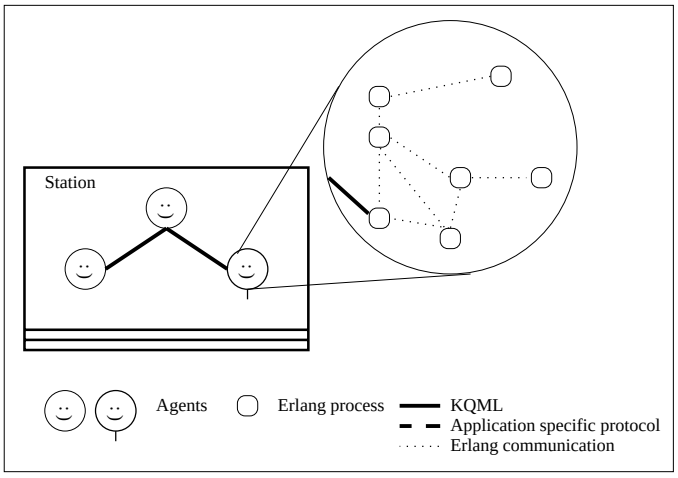

Figure 2: Relation between agents and ERLANG *processes*

For the agent specific functionality we choose to stay with ERLANG as the agent programming language, instead of implementing a new special-purpose language. Agent-specific functionality is provided through special functions, which is the normal way to add functionality to ERLANG.

First of all we need a set of libraries to solve the requirements for security:

- Secure communication
- Authentication
- Resource handling

These will be described in more detail below. Then we identified a number of obvious agent functions, much like the cooperating system for agents described in SodaBot[3]. Agents

typically have to interact with other agents and users, write and send email, and access local information. Remember that one of the requirements was the ability to exchange sensitive and confidential information. In the next section we will describe functions for:

- KQML
- Email and file system
- Visualization

KQML

KQML[4] is used as the protocol for agent communication. The protocol has existed some years now, and there are several applications that use it. We have nothing to say about whether KQML is the most suitable protocol for our application or not, but we have used it since it is a de-facto standard, and we want to be able to communicate with agents running inside other agent systems.

We see three types of bidirectional communication in the system:

- station to station
- agent to agent
- agent to user

Our approach is to use KQML for all inter-agent communication, whether it is between agents in STEAM or between a STEAM agent and an external agent on an external agent platform.

Station-to-station communication is actually communication between two station masters, a special kind of stationary agent, and thus uses KQML.

Communication between an agent in STEAM and a user interface also uses KQML – the agent sees its user interface as another agent.

The communication between agents is location transparent thanks to the distributed name server in the communications layer which at all times keeps tracks of the current location of an agent.

The STEAM system uses a number of different transport protocols, depending upon whom the system is communicating with. Internally, between STEAM stations, distributed Erlang is used, while TCP/IP is used to communicate with external agents. Adding more transport protocols is transparent to the existing agents.

KQML implementation We have implemented a subset of KQML which follows the syntax and semantics specified by Labrou[12].

We have only implemented those parts that were useful in our project, i.e. performatives for questions, responses, and problem reports.

STEAM does not implement a *KQML facilitator*. The name server functionality is implemented in the communications sublayer. The broker functionality has been simplified and is implemented as a simple *Service Lookup Agent* (SLA) instead.

Service Lookup Agent Each station has an SLA. Agents may register their *agent names* with the SLA or the *services* that they can perform. An agent looking for another agent, either by its name or by a service that it can perform, asks the local SLA for the address of a fitting agent. The agents then communicate directly.

Secure communication

To achieve a secure communication between agents a slightly modified version of the Secure KQML protocol, proposed in [17], has been implemented.

Communication security is based on the notion of *secure domains*. In a secure domain, all stations are assumed to be friendly, and secure communication is possible without encryption. This is the case in a company network behind a firewall. Communication with stations outside the domain is done by encrypted connections. A domain may consist of one or more stations.

The security enhancement is added to the KQML implementation as an agent sub layer without involving the agent application. The agent application specifies the desired degree of security to a crypto-aware Security Manager. Through a handshake with the receiving party it secures the messages up to the desired level.

The implemented Security Manager supports privacy, authentication, integrity of messages, non-repudiation of message origin and protection against message replay.

The cryptographic and signing facilities are implemented as library calls using existing RSA implementations in C[15].

The Secure KQML protocol consists of two phases: the authentication phase and the privacy phase. In the authentication phase initial parameters for a private conversation are setup. During the privacy phase the actual messages are sent and received.

Performance improvements

Achieving privacy by using RSA to encrypt the whole messages is quite demanding, computation wise.

The approach in PGP[19] and SSH[18] is more efficient as only a small session key is encrypted by RSA. The rest of the message is encrypted with the session key using symmetric encryption methods such as DES and IDEA. Only lack of time has prevented us from doing the same; STEAM could easily be extended to use the more efficient method.

Authentication

Agents must be sure of the identity of the party that they wish to exchange information with. For this a two-way authentication protocol is used which is based on RSA authentication using 256-bit random strings.

Each agent is given two pairs of RSA keys, one to sign messages and the other to encrypt. The public keys are distributed in the system through a distributed database, and are available to all agents.

Resource handling

To control the usage of resources in the system a scheme similar to Teleclicks in Telescript was implemented. In STEAM, the unit of resource consumption is called a *metapill*.

Each agent has a finite supply of metapills, which is decremented each time the agent uses some system service. The agent is also charged for usage of memory and CPU-time.

The purpose of the system is primarily to stop agents from using all the resources at a station (and thus denying other agents service). If the system is connected to an electronic payment system, it would allow agents to bill each other for services.

Email and file system

Agents may need to use email to communicate with users, or to access the data stored in the local file system. In order not to violate security, this is performed through stationary agents. Mobile agents cannot directly access any local resources, or send email. It was not necessary to implement new functions for this, as adequate support for email and file system access is provided in standard ERLANG libraries.

Visualization

The visualization part can be considered as a separate system. It can in fact be used by another client or run separately with some modifications. The visualization acts as a separate agent connected to the STEAM station. The visualization system is used in two ways: to monitor the steam station and to implement the user interface in the application (our meeting scheduler).

Communication between the client system (in this case a STEAM station) and the visualization package is done using KQML over TCP/IP. A conversation is established between a client application (e.g. station master) and an applet specially designed for the application (e.g. showing the action taking place on the station). Information is then sent between these two via the visualization master in the client system and its correspondence in the visualization system.

The system is completely implemented in Java for portability reasons, and the choice of TCP/IP was to make the visualization system flexible. This makes it is easy to replace the visualization in the STEAM system or to use it in a different system.

2.5 Application: a meeting scheduler

The overall architecture for an application consists of a set of stationary agents having access to system specific resources and a set of mobile agents which act on behalf of the stationary agents. The interaction between agents and the user is performed via one or more user interface programs (applets or something else). The user interface programs look like ordinary agents to the application agents.

The application we chose to implement was a meeting scheduler which was used to validate the requirements of the STEAM system. The design and implementation was more focused on validating the requirements than fancy features.

The meeting scheduling system is composed of three separate systems, each with its own autonomous functionality:

- A user interface that can supply information about a meeting to be scheduled.
- An existing personal calendar system.
- An agent appointments system that can schedule joint meetings between several participating users and locations.

The user interface is a Java applet running in a web browser. The existing calendar system is Ical[6] that offers all the functionality expected from a personal calender system. It has a nice and easy to use X based user interface and stores the appointment information in a simple and easily extensible file format.

Each user has two agents: a stationary master appointment agent residing on the user's home station, and a mobile slave appointment agent. Locations (meeting rooms) also have a pair of agents but these can reside on any STEAM station. Each group of users and locations that can use the appointment system has their appointment agents registered in a group specific name server agent. Both users and locations can belong to several appointment groups at the same time. One or more constraint solver agents are used to solve the problem of finding a time interval in which all participants have nothing scheduled.

The meeting scheduler system can schedule a meeting given a list of participants, a date, a location, and a time range within which a meeting of a certain duration is wanted.

2.6 An example

A user, Sue, wants to set up a meeting and opens the appointments system applet in her web browser. Sue wants to make sure that no one else is able to set up meetings in her name so she has made this applet private. This means that she has to authenticate her identity by entering an encryption key before continuing.

The applet then sets up a connection to Sue's master appointment agent and it responds by sending back a list of possible participants and locations to choose from, i.e. those currently registered in the appointments system's name server.

Next, her master appointment agent tells the applet to show a form where Sue fills in the data for the meeting. Sue selects the participants `Tom@beach`, `Ken@boat`, and `Cecil@beach` and `Cecil@beach`. She estimates the meeting duration to 2 hours and specifies that it must take place February 11, at some time between 9 AM and 3 PM. The meeting location is selected to be room 34:1 and the subject for the meeting is "the STEAM project". When Sue is finished she presses the send button and lets the appointment agent system take over.

The stationary master appointment agent on Sue's STEAM station receives a request to book the new meeting. Its first task is to contact the master appointment agents of the participants and sends them a request specifying the data of the meeting. All participating master appointment agents extract the time slots available on the given day from their Ical calendar files. They then forward the information to their mobile slave agents. The slaves then travel to the station where the constraint solver agent resides.

When all participating slave agents have arrived at the constraint solver agent, Sue's slave

acts as an administrator. It then makes sure that all data from the other slave agents are sent to the constraint solver, which in turn then tries to find a suitable time. The result from the constraint solver is sent back to the administrator (Sue's slave) which then forwards it to the other slaves.

If a joint meeting could be scheduled the slave agents return back to their home stations with the result. The master appointment agents then update their respective Ical calendar files with the new appointment and notifies the user by sending them an email about the new appointment. Sue is notified via the Java applet instead. If no meeting could be scheduled Sue is notified about it via the applet, and she can try again with new data for the meeting.

3 Conclusions, related and future work

We found that ERLANG was fairly well suited for programming an application with secure mobile agents. Although many essential things such as encryption, KQML, etc. are missing from ERLANG, this functionality can be added through a set of library functions.

The main problem was that distributed ERLANG had to be used since the name server uses the distributed database Mnesia. This caused problems with the security between stations. One station can for example halt any station within the agent system due to the distributed ERLANG philosophy. To prevent this the ERLANG language itself must be changed and thereby disable much of the features which distributed ERLANG offers.

However, as a workaround we defined secure domains, where distribution is allowed, and all communication to other domains is done by encrypted connections. Further work has to be done to decide which algorithm to use to spread trusted domain addresses. One possible solution is to use the PGP model[19].

Our application could perhaps have been better chosen had mobility been an essential demand. However, there are advantages even for a simpler application such as our meeting scheduler in using mobile agents[9].

Related work

Many systems are being released which support mobile agents. Telescript[16] is an environment that supports portable and secure code. The programming language is object-oriented and provides an operator that can move running programs with its state between machines during execution. Telescript has also focused on security with with encryption and digital signatures. Our approach and also AgentTCL's[8] is in many areas similar to Telescript. However, our system is based on a high level symbolic functional language, where we with a set of minor adjustments are able to satisfy our requirements on security for mobile agents.

A lot of activity can be seen around Java[7], due to its incorporation in several Web browsers. Java per se is not a system for mobile agents, but mobile agents, but with extensions like MOLE[2], Aglets[13], JavaAgent[5] etc. you can build such a system. The security though seems inflexible: the message "security violation" pops up as soon as you try to do something useful.

Java vs. Erlang

As the project was using both Java and Erlang it is natural to compare the two languages, although they were used for different purposes. Java was mainly used for the graphical interface, but code was also written to parse and generate KQML. The execution performance of Java was much lower than we had expected, especially in the parser, which we could compare directly to the corresponding code in ERLANG.

To us it seems the garbage collector in our version Java for SUN (JDK 1.0.2) was one of the main reasons for this inefficiency, but the implementation of the Java Virtual Machine was probably also not particularly efficient. Some bugs in the Java development environment indicated that the product is a bit immature, but nothing that prevents it from being usable.

Java (in JDK 1.0.2) uses cooperative multitasking, where each thread must give up its execution voluntarily. This complicated programming. ERLANG on the other hand has full preemptive multitasking, and seems to handle a large number of processes much more efficiently. Both languages are nice to work with, although the code in ERLANG is more compact and readable.

Further work

Although ERLANG has some shortcomings as an agent programming language it seems better suited than most alternative approaches.

One aspect that should be improved is the fact that there is no way to really migrate processes in ERLANG. The only way to achieve this now is to spawn new processes in the new runtime system and kill the old processes. We intend to investigate the possibility of extending the ERLANG run time system so that a group of processes may be explicitly moved to another node.

We would also like to add basic security support to the ERLANG run time system, based on results from this work and work on SafeErlang[14].

Mobile agents are particularly advantageous in systems where it is hard or impossible to maintain a connection between the nodes, such as in radio-based systems. An agent with a complicated task need be sent just once, and this will reduce the traffic on the connection considerably. This is particularly important in the case when the user agent is simplistic, and requires a lot of interaction. A promising field for exploration is therefore the use of mobile agents for implementing advanced functions in PDA:s and other portable terminals, such as mobile phones. These systems are a challenge as they have limited memory and computing capacity, and the security requirements are vital for most applications.

Bibliography

[1] Armstrong, J. L., Williams, M. C., Wikström, C. and Virding, S. R., *Concurrent Programming in* ERLANG, 2:nd ed. Prentice Hall, 1996.

[2] Joachim Baumann, *Mole—A Java based Mobile Agent System*, IVPR Stuttgart, 1996.

[3] Michael H. Coen, *SodaBot: A Software Agent Construction System*, MIT AI Lab, USA, 1995.

[4] Tim Finin et. al., *Specification of the KQML Agent-Communication Language*, The DARPA Knowledge Sharing Initiative, 1993.

[5] H. Robert Frost, *JavaAgent Template*, Stanford University, EIT, 1996.

[6] Sanjay Ghemawat, *Ical user guide*, sanjay@lcs.mit.edu, `http://www.pmg.lcs.mit.edu/~sanjay/ical.html`.

[7] James Gosling, Henry McGilton, *The Java Language Environment*, Sun Microsystems, 1996.

[8] Robert S. Gray, *Transportable Agents*, Ph. D. Thesis, Dartmouth College, 1995.

[9] Colin G. Harrison, David M. Chess, Aaron Kershenbaum, *Mobile Agents: Are they a good idea?*, Research Report, IBM Research Division, T.J. Watson Research Center, 1995.

[10] Henry A. Kautz, Bart Selman, Michael Coen, *Bottom-Up Design of Software Agents*, Communications of the ACM, July 1994.

[11] Frederick Colville Knabe, *Language Support for Mobile Agents*, School of Computer Science, Carnegie Mellon University, USA, 1995.

[12] Yannis Labrou. *Semantics for an Agent Communication Language*. A Doctoral Dissertation for the PhD Defense Examination Submitted to the Defense Committee at the Computer Science and Electrical Engineering Department (CSEE), University of Maryland Graduate School, Baltimore, Maryland, 21228-5398, September 10, 1996.

[13] Danny B. Lange, *Java Aglet Application Programming Interface*, IBM Tokyo Research Laboratory, 1996.

[14] Gustaf Naeser, *Erlang and Mobile Agents*, M.Sc. Thesis, Computing Science Department, Uppsala University, 1996

[15] RSA Laboratories, *RSAREF(TM), A Cryptographic Toolkit*, Revised March 25, 1994, Version 2.0

[16] *Telescript Technology: Mobile Agents*, General Magic, USA, 1996.

[17] Chellia Thirunavukakarasu, EIT, Menlo Park, Tim Finin and James Mayfield, Computer Science and Electrical Engineering, University of Maryland Baltimore County, *Secret Agents—A Security Architecture for the KQML Agent Communication Language*, USA, 1995.

[18] Tatu Ylönen, *Secure Shell*, `http://www.cs.hut.fi/ssh/`.

[19] Philip Zimmermann, *The Official PGP User's Guide*, MIT Press, 1994.