Applying SICS: A technique for software modules Communication.

T. Spyrou[1], E. Sutinen[2], A. Douma[1], H. Putkonen[2], J. Darzentas[1]

[1]University of the Aegean
Research Laboratory of Samos
GR 83200 Karlovassi, Samos
tsp@aegean.gr

[2]University of Oulu
Tol, Oulu
FIN 90570 Finland
putko@tolsun.oulu.fi

Abstract

In this paper SICS, a message parsing and message communication system, is presented as an effective means of intermodule communication during distributed software implementation of complex systems. The use of this communication system is shown through SECURENET, one of the major RACE research projects in the area of security of management in network environments. SICS plays a basic role in SECURENET since it is the message passing system between the different modules. The application and use of SICS is presented here through its use by UII one of the SECURENET modules which is responsible for detection of malicious attacks. Also a number of experiments have been carried out to evaluate the characteristics of SICS and to test its potential in fast message transferring under varying conditions. SICS plays the role of communication manager and in parallel it has the responsibility for maintaining a common communication dictionary. In this way the rewriting of code is minimised during implementation and this is a major advantage in the development cycle.

Keywords: Intermodule communication, software modules, software architecture.

1. Introduction

This paper presents the use of an approach which is used to improve efficiency during the implementation of software modules. This approach is considered as a prerequisite when various software modules need to communicate during implementation. The problem of software modules' communication is discussed and the actual system, SICS [6,7,8] (SECURENET Internal Communication System) is proposed as a possible message passing system which solves a number of such communication problems.

SICS is also considered as a way of minimising the rewriting of code during implementation offering in this way a major advantage in the development cycle.

The initiative for building SICS comes from LINDA [1,2,3] a system for dividing software tasks among several processors.

The main application of SICS till now is its use in the SECURENET project [5]. The SECURENET project is developing a network monitoring and analysis system which aims at networks protection. It is composed of several modules with different responsibilities. There are currently three analysis modules which analyse information coming from the actual network in order to recognise or to infer malicious attacks on the network. There are several monitoring modules which observe the real network acquiring the information necessary for the analysis step. There is a decision support module which has the responsibility to elaborate the results of the analysis made by the analysis modules. All these modules need to pass information to each other in an efficient way. SICS is used to play the role of communication manager and in parallel it has the responsibility to maintain a common communication dictionary.

The application and use of SICS in SECURENET project is also described. SECURENET [5] is one of the major RACE research projects in the area of *security of management* in network environments. SICS plays a basic role in SECURENET since it is the message passing system between the different modules. The application and use of SICS is presented through its use by UII [4] one of the SECURENET modules which is responsible for detection of malicious attacks. Also a number of tests was carried out to evaluate the characteristics of SICS.

For the purposes of these tests, communication has been established between an agent that generates messages and the I/O function of UII module which is receiving these messages, processes them, and returns output messages to a message receiving module.

Based on this configuration, a set of experiments have been designed and performed. Special attention was given to the speed of message processing and transmission in relation to their size and the complexity of their structure.

Finally the advantages of using SICS in SECURENET development cycle is discussed.

2. Overview of SICS

SICS (SECURENET Internal Communication System) is a message parsing and communicating system which was initially designed and built for use in the SECURENET project. The main aim was to provide all the necessary organisation and support to the rest of the software modules of the SECURENET project to cover their communication needs. Currently SICS is an integrated communication system which can cover communication needs wherever special requirements exist.

SICS is a set of conventions, services and methods and their implementation, which allow efficient communication between modules of a system and automatic parsing of messages. The main tasks of SICS are:

- to pass messages between the modules,
- to manage the connections and the data used in communications,
- to launch objects dynamically when needed
- to manage data in messages independently of the processor architecture
- to configure the system behaviour even at runtime and
- to make the system state persistent.

2.1. Specifications and design

The main primitives of the SICS system are:

Message which is what is actually moving in SICS.

Object which means any program or function that is running and talking to other objects and doing its part for SECURENET.

Class is a template for program or function that can be started. Inside SICS each class also has information concerning its launching procedures.

Channel is an abstraction of data streams from objects with all the required information for connections from queues and based on opening inactivated queues, to activate objects at the other ends. Channels are templates for connections. Channels are handled by the default handler.

Connection is a logical data stream between one or more objects. Connections are seen by the communication server and configured there by the default handler.

Queue is the part of connection that an object sees.

Protocols are definitions of data structures using which information is packed into messages and sent down the channels.

Configuration files are text files that contain the configuration of many areas of SICS. They are converted into binary data structures for fast use.

Objects and communication server discuss using *control messages.*

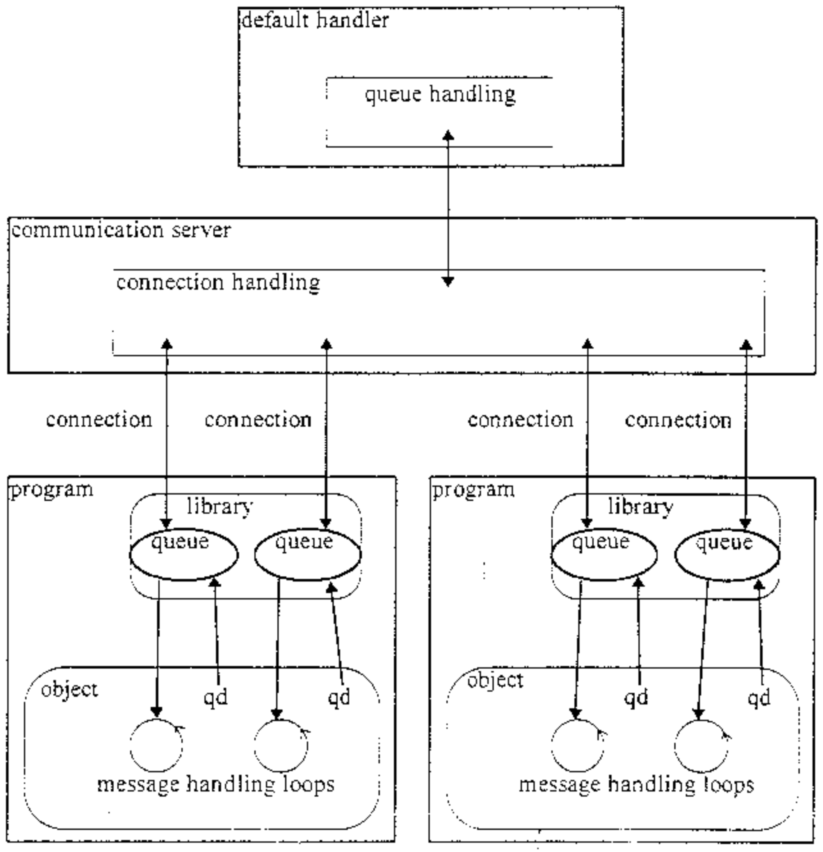

Figure 1: Structure and terms

Figure 1 presents a view of the terms in their context. In this figure term 'qd' refers to queue descriptor and program to a physical host of a running object.

SICS consists of the following parts:

- The Communication server which is the main part of SICS and handles messages passing from objects to others and informs the default handler of every situation for which it has not been configured.

- The Shared memory which is the media of communication between objects. It has a formatted control message area that is used for communication between the communication server and objects.
- The Communication library which consists of a set of functions, variables and data-areas that handle communication between communication server and objects. The communication library includes protocols' configuration and the functions to access data in messages.
- The Default Handler which maintains structures to contain the situation inside running the SECURENET system based on those structures. It configures the communication server and also activates and deactivates objects when needed. The default handler reads configuration files and then takes actions based on them.
- The Basic objects which give basic services to other objects and give direct administrational access to SICS.

SICS has been divided into two parts; the server and the default handler. SICS acts as a fast packet switch and the default handler takes care of the brainwork of opening connections when needed, starting new objects etc.

When SICS is started, the server is first started. The server initialises shared memory segment and after that starts the default handler. From then on the server maintains connections, passes messages from clients to other clients based on connections and handles the memory management of the shared memory block. After the default handler has been started, it waits for connection requests. The default handler starts client objects when needed.

When an object wants to send something to other client, it creates a connection with the server using library functions. A packet is assembled by the object and sent to the queue. The server then gives this packet to client.

Next, the server passes all messages on channels which are not currently open to the default handler. The default handler starts the object on the other end of the channel if appropriate and creates the connection, and sends the packet back to the server. Now when the connection has been made, the server can forward all subsequent packets on that channel directly to their correct destination.

2.2 Implementation issues

Initially, SICS was implemented using the Linux operating system on normal PC - compatible hardware. The development tools are listed below:

• C++ compiler:	GNU g++ v. 2.5.8
• Debugger:	GNU gdb v.4.12
• Editor:	GNU emacs v.19.22

SICS has been implemented according to POSIX standards, and now runs on several platforms that support GNU tools. It is relatively easy to port to any system which uses SYS V style shared memory and UNIX domain sockets. Further development of SICS eventually helps porting it to other operating system platforms.

Shared memory is allocated by communication server. It has five areas: version area, queue to communication server, queues to objects, message slot table and data area.

The version area is only a few bytes long at the very beginning of the shared memory. It contains data about the version of the communication server. This means the version of the queues that are after it in the shared memory. Using that information clients can check if they do understand this version and either terminate without causing havoc or use the appropriate message handler.

Queue to communication server is a data structure that contains a message queue as well as lock and queue pointers that point to last read and written slots in message queue. Each message slot contains sender's id, message type and four parameters.

Queues to objects represent connections/queues of objects. They contain one slot for each connection. Each message slot contains a lock, time stamp of latest communication and pointer to actual message slot in message slot table.

Message slot table is an array of structures that contain control messages to objects. Each message can go to several objects, but the slots in this table do not contain pointers to objects that are reading them, instead the queues to objects have pointers to message slots. Each slot in this table contains messages sender's id, message type and four parameters. Control messages of type MSG_MSG contain pointers to message table.

Data area is most of the shared memory. It is allocated message areas by the communication server to be used by objects in communication.

Note that the version area and queues and message slot table do not need to reside in the same physical shared memory area as the message area. This helps debugging programs that reference memory incorrectly, and if fixing them fails, provides a slow but workable method to connect them into SECURENET.

The communication server is divided into the main loop, the message parser, memory management and connection management.

The communication server is the first program that starts when SICS starts. Upon start up it accesses shared memory, initialises several variables and launches the default handler. After that communication the server goes into the main loop.

The main loop has several steps:

- it checks if there are any incoming messages to the server and if any, it hands that message to the message parser. Note that messages that the communication server receives are control messages.
- on idle time it checks if any slots in the message slot table have an expired time stamp and resolves the situation.
- at pre-set times it sends gathered statistics of systems operation to appropriate objects if necessary.

3. A trial application

For the purposes of this paper, SICS system has been checked through its use during the implementation of UII module. UII module is one of the basic modules of the SECURENET system, a system for the detection of malicious attacks against the management of a network. Next section gives an outline of the general problem environment describing the modules of SECURENET and their intercommunication.

3.1 The problem (SECURENET)

The main application of SICS till now is its use in SECURENET project. SECURENET project is developing a network monitoring and analysis system which aims at networks protection. It is composed by several modules with different responsibilities. There are currently three analysis modules which analyse information coming from the actual network in order to recognise or to infer about malicious attacks to the network. There are several monitoring modules which observe the real network acquiring the information necessary for the analysis step. There is a decision support module which has the responsibility to elaborate the results of the analysis made by the analysis modules. All these modules need to pass information each other in an efficient way. SICS is used to play the role of communication manager and in parallel it has the responsibility to maintain a common communication dictionary.

3.2 SICS in SECURENET

Using dedicated message passing system like IPC and sockets instead of standardised network and interprocess communications methods may seem inappropriate. One might easily think that system like SICS is just extra overhead which does not offer true value over generic. However, its advantages in development cycle are very clear. Rewriting of code is minimal, because lot of work in communications is done automatically, via configuration files settings. If a change in the agent's output to another agent is required then the only requirement is the corresponding edit to the configuration file and no code rewrite is required.

3.3 The specific implementation

The User Intention Identification module, as mentioned above, plays a complementary role within the SECURENET system, trying to detect a range of malicious attacks in a network. According to the SECURENET architecture the specific module needs to *get* audit data from a module that monitors the network activity, to analyse them and then to *send* the results of this analysis to a decision module. SICS is used in order to offer an effective communication between these three modules.

In order to communicate using SICS, a number of definitions have to be done and a set of SICS functions have to be used. These necessary definitions and the required functions are described in the sequel.

First of all, the classes or objects that participate to the communication must be declared in the *classes* configuration file. For the purposes of this work the specific objects are the *logagent* object that sends the audit data and the *UII* object that receives them and sends the results for further processing. A sample description of these two objects follows.

```
// Name:          logagent object
// Description: Logagent class to handle log message storage
&class logagent
     {
          ProgramName = SICS_logger;
          FileName = "SICS/bin/logger";
          Type = PROCESS;
     }

// Name:          UII object
// Description: The User Intention Identification class
```

```
@class UII
      {
            ProgramName = SICS_uii;
            FileName = "UII/UII";
            Type = PROCESS;
      }
```

All configuration files are written in ASCII format. Due to performance issues these are converted to binary files. All text on a line after # or // is considered as comment text.

For each class the class name, the class process name, the executable file name and the process type are specified. The "file name" field determines the program to be started when the specific class sends or receives a message for the first time. The "process type" field specifies if the program code is a separate program or not.

When an object is designed, a set of channels, which will be the connections, must be designed too. Thus, for the specific module two channels are declared in the corresponding configuration file, the channel AUDIT where audit data are transferred

```
// Name:          AUDIT
@channel AUDIT <60>
{
      Type = BUS;
      Writer = logagent;
      Unique = YES;
}
```

and the channel UIIMSG where the results are forwarded:

```
// Name:          UIIMSG
@channel UIIMSG <61>
{
      Type = BUS;
      Writer = UII;
      Unique = YES;
}
```

The "type" field specifies the way the objects send or receive the messages. Possible values are *to_many* when one object writes a message and many objects receive; *from_many* when many objects write and one receives; forward; backward; bus; and bidirect. The "writer" or "reader" field specifies the name of the object that reads or writes a message from/to the specific channel.

Each object can send or receive information via a channel, using a special protocol. In other words, protocols are the actual way objects pack data into messages and communicate them. Protocols' configuration file contains definitions for each protocol used. For the current application two protocols for messages are defined:

```
$protocol AUDIT_REC : SECURENET <30, 1>
{
     long   nodeid;
     long   time;
     long   recid;
     long   system;
     long   event;
     long   returnc;
     long   errno;
     long   pid;
```

```
        long   ppid;
        long   auditid;
        long   realid;
        long   effid;
        long   grip;
        long   effgid;
        long   syscid;
        int    nparams;
        string sysid;
        string syscall;
        string params;
        }

    @protocol UIIMSG_REC : SECURENET <31, 1>
    {
        string sysid;
        long   node_id;
        long   time;
        long   recid;
        long   user_id;
        long   hyptype;
        long   conflev;
        long   nparams;
        string params;
```

Protocols are stored as a set of fields in a specific order. Each field denotes the type and length of each piece of data that can be stored into a message. These definitions are converted by a protocols' converter program into a binary array of structures, These arrays are directly transferred into memory and they are used to parse messages very fast. Protocols' converter also checks the syntax of that file and creates *C-format include* files which contain macros for even faster data retrieval/storage.

When an object wants to send information to another object, it establishes connection with SICS using SICS library functions. The corresponding packets are assembled by the object and are sent to SICS queue. SICS then gives these packets to receiving objects.

Even if this process seems complex, in fact, it is realised by just moving pointers in shared memory. No data is actually moved while messages are transferred from senders to receivers.

According to the normal way objects utilise SICS, the following steps describe the implementation of the communication of UII module with the other modules:

- call SICS initialisation

```
    if(SICS_init_sics() == -1)
        {
        printf("Could not initialise SICS service. EXIT \n");
        exit(0);
        }
```

- call SICS channel-queue initialisation

```
    auditqueue=SICS_open_queue(SICS_CHAN_AUDIT, SICS_QTYPE_NORMAL,
                                          uiihandle_msg);
    uiimsgqueue=SICS_open_queue(SICS_CHAN_UIIMSG,SICS_QTYPE_NORMAL,
                                          uiihandle_msg);
    if ((auditqueue < 0 )  |  (uiimsgqueue < 0))
```

```
          {
          fprintf(stderr, "%s: Initialisation of queue failed\n", SICS_progname);
          exit(0);
          }
```

- Do an infinite loop that handles all incoming messages, sends messages to channel and gives time for other programs

```
          while(!SICS_do_exit)

                    while(SICS_loop_once(&SICS_buff[0]) != -1)
                    ifdef DEBUG
                         if(SICS_buff[0] != '\0')
                              printf("%s\n", &SICS_buff[0]);
                    endif
          /* Call your function that sends messages */
                    sender();
          /* Give time for other programs */
                    SICS_usleep(seconds);
               }
```

- close all queues and call SICS quit function

```
     SICS_quit_sics();
```

The *sender* function checks if the corresponding channel is available

```
     if(SICS_check_queue(uiimsgqueue))
               return(0);
```

then sends the message

```
     SICS_send_UIIMSG_REC(message, uiimsgqueue, 0);
```

and recycles the reserved memory

```
     SICS_recycle_text(message);
```

Finally there is the *handler* function "uiihandle_msg" which is called from SICS whenever there is an incoming message. In this function only a pointer that points to the received protocol is defined :

```
     static int uiihandle_msg(struct SICSPS_ *msg, char *buf)
     {
          struct SICSPS_AUDIT_REC *message;
          if(msg->protocol != SICS_PROT_AUDIT_REC)
                    return 0;
          message = (struct SICSPS_AUDIT_REC *)msg;
     /*Call a function that needs the received message*/
          ext_function;
     /* Free the memory */
          SICS_recycle_text(message);
          return(0);
     }
```

This implementation has been done using C paradigm. SICS offers the possibility of using C++ instead of C but both these two ways are based on the same basic idea.

4. Design of the experiments-Measurements

To estimate the performance of SICS in message passing operation, the C programs implemented for the communication of UII were modified in order to perform a set of measuring experiments. The program "send" sends messages and the program "receive" counts the amount of messages received per second and sends this information back to "send" which returns the actual benchmarks.

For the experiments a pentium based machine with 16MB of main memory running LINUX operating system was used. The message transfer rate was evaluated in relation to the message length and the complexity of the message structure.

The results are shown in the following three figures.

Figure 2 shows the message transfer rate in relation to the length of the transferred message. Messages with twenty different length values have been chosen. The length values vary from 100 to 2100 characters. For each one of these values a big number of message transfers have been performed and the corresponding average speed has been measured.

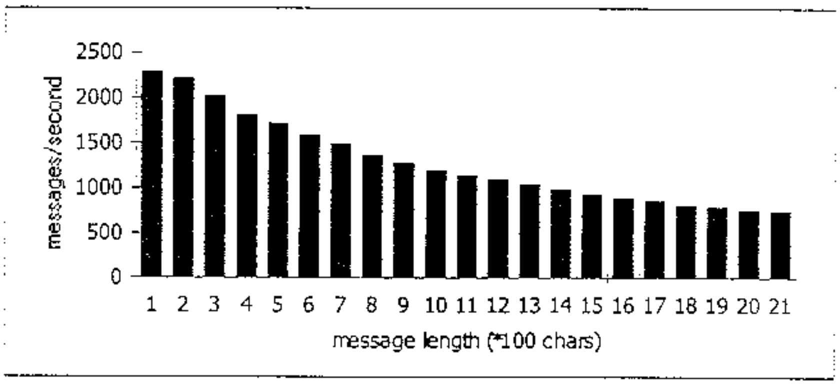

Figure 2. message transfer rate in relation to the message length

The main conclusions that arise by observing the above figure are:

The message transfer is very high, almost 1000 messages per second even if the length of the message is quite big, 2100 characters.

The message transfer rate decreases as the transferred message length increases.

For the same experiment the rate of change $\left(\frac{\text{change of transfer rate}}{\text{change of message length}}\right)$ has been calculated and is shown in figure 3.

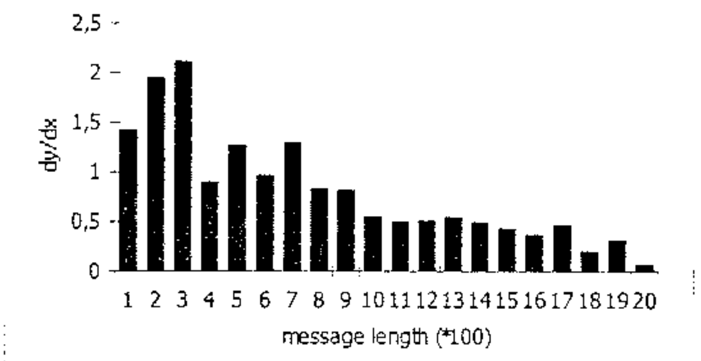

Figure 3. rate of change in message transfer in relation to the change in message length

The rate of change decreases as the message length increases.

For a constant message length of 500 characters, seven experiments have been planned in order to measure the change in message transfer in relation to the complexity of the transferred message structure. Figure 4 presents the results of these experiments.

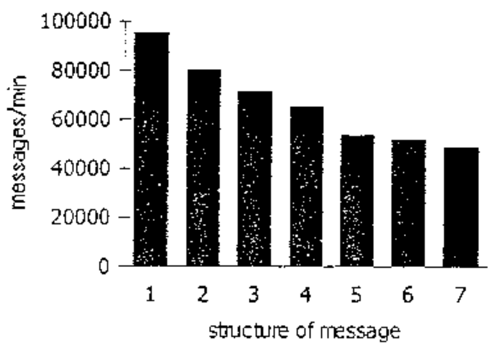

Figure 4. message transfer rate in relation to the complexity of message structure

The seven structures selected are described in the following table:

Experiment	Structure of message (# x field length = total length)
1	1 x 500 = 500
2	2 x 250 = 500
3	2 x 200 + 1 x 100 = 500
4	3 x 100 + 1 x 200 = 500
5	5 x 100 = 500
6	2 x 50 + 4 x 100 = 500
7	4 x 50 + 3 x 100 = 500

Table 1. Structure of the transferred messages.

The results of this experiment show that the message transfer speed is a function of the complexity of the structure of the message to be transferred.

However for systems such as SECURENET, where the actual processes which are carried out by the software modules, the speed is by far greater than the requirements of the modules. This applies even for the accumulation of the modules requirements. That is, each module can process maximum 100 messages per second, and it has been shown that SICS can process very comfortably about one thousand message per second in the most demanding case.

5. Discussion and Conclusions

In this paper SICS, a message parsing and message communication system, has been presented as an effective way of intermodule communication during distributed software implementation of complex systems. The use of this communication system has been shown through its use in the development of UII one of the SECURENET's modules. Also a number of experiments have been carried out to test its potential in fast message transferring under different conditions.

SICS is used to play the role of communication manager and in parallel it has the responsibility to maintain a common communication dictionary. In this way the rewriting of code is minimised during implementation being a major advantage to the development cycle.

Using dedicated message passing system like IPC and sockets instead of standardised network and interprocess communications methods may seem inappropriate. One might easily think that systems like SICS is just extra overhead which does not offer true value over generic. However, its advantages in development cycle are very clear. Rewriting of code is minimal, because a lot of work in communications is done automatically, via configuration files settings. If a change in an agent output to another agent is required then the only requirement is the corresponding edit to the configuration file and no code rewrite is required.

6. References

[1] S. Ahuja, N. Carriero, D. Gelernter and V. Krishnaswamy, "Matching Language and Hardware for Parallel Computation in the Linda Machine", *IEEE Transactions on Computers*, August 1988, Vol 37, No 8, pp. 921-929.

[2] L. Cagan and A. Sherman, "Linda unites network systems", *IEEE Spectrum*, December 1993, pp. 31-35.

[3] N. Carriero and D. Gelernte "LINDA in context", *Communications of the ACM*, April 1989

[4] J. Darzentas, T. Spyrou, et. al. Functional Specifications of SECURENET Components, *In: SECURENET System Development Plan, CEC RACE Report, R2057.EXP.DR.L.050B1*, pp. 53 - 143, 1992.

[5] P. Spirakis, S. Katsikas, D. Gritzalis, F. Allegre, J. Darzentas, C. Gigante, D. Karagisnnis, P. Kess, H. Putkonen and T. Spyrou "SECURENET: A Network Oriented Intrusion Prevention and Detection Intelligent System", *Network Security Journal*, vol. 1, no 1, Nov. 1994. (Also in IFIP SEC94, *Proceedings of the 10th International Conference on Information Security*, May 1994.)

[6] E. Sutinen, "/DEL/IMP/4.2/OULU/EPS/050395; Implementation of SICS", *Securenet II deliverable*, 1995.

[7] E. Sutinen and H. Putkonen, "/DEL/SPC/4.1/OULU/EPS/050594; Specification of SICS", *Securenet II deliverable*, 1994.

[8] E. Sutinen and H. Putkonen, "/TRP/DES/4.2/OULU/EPS/050395; Design of SICS", *Securenet II deliverable*, 1995.