Protecting Co-operating Mobile Agents Against Malicious Hosts

Regine Endsuleit and Thilo Mie

Interner Bericht 2002-8

Institut für Algorithmen und Kognitive Systeme
Universität Karlsruhe (TH)
WS 2002/2003

ISSN 1432-7864

Abstract. We propose a security model for open multi-agent systems. Given a user-defined task T, we generate a set of mobile agents which realize a common functionality that solves T. Those agents co-operate with each other and build an autonomous community. Using a scheme for secure distributed computations, this community is able to perform secure computations without requiring interaction with a trusted party. For this paper, we have chosen Canetti's model for secure multi-party computations (see [Can01]). Unfortunately, the problems arising from the migration of agents are not covered by this technique. We present an extended model that offers a solutions to this. Thus, we yield guarantees for confidentiality of secret data, detection of unauthorised code and data changes, reestablishment of corrupted agents and prevention from malicious routing.

1 Introduction

Mobile agents are designed to roam the network autonomously and have their code executed by foreign hosts. (Details about multi-agent systems can be found in [Wei99].) They are the consequent answer to our growing networks as well as to the user's need to collect, filter and process huge amounts of information even though his bandwidth or computing resources might be limited. The broad range of applications includes mobile computing, information retrieval in large repositories and e-commerce applications like price negotiations. But until now there is no general solution to the security problems in such open multi-agent systems. Some authors (see e.g. [ST98],[LM99]) worked on a technique called "function hiding" to achieve confidentiality of the computation and protection against software-piracy. In a function hiding scheme a sender A encrypts a function f he wants to be executed by a second party B. Then B evaluates the encrypted function $E(f)$ on his input x. The result $E(f)(x)$ is returned to A and decrypted by him, yielding the result of $f(x)$. Yet, the published approaches can only hide limited classes of functions.

A different approach is to obfuscate the code, in a way that the functionality is preserved, but nobody can see how it works. As anyone knows, it is hard to understand source code. Motivated by this, Hohl suggests in [Hoh97] to mess up source code by the use of insane variable identifiers and completely unstructured implementation to make it even harder to comprehend. Unfortunately, the readability can be improved by compilation and subsequent de-compilation.

Most of the preceeding research aimed at the construction of an obfuscating compiler because the availability of an efficient method for obfuscating programs is very important for the use of mobile agents. A final point to this research was reached in October 2001 when Barak et. al. (see [BGI01]) proved that the existence of such a compiler is impossible as long as one requires the resulting program to have a virtual black box property. This result does not mean that any research in function hiding is obsolete. There is still hope to find function hiding schemes like homomorphic encryption, but they would not offer an efficient way to construct an obfuscating compiler.

It seems to be hard to secure stand-alone agents. Therefore, we diverge from this and consider a completely different approach. By using a community of collaborating mobile agents, it is possible to increase reliability of the community's functionality by mutual control.

Roth suggests this idea in [Rot99] by launching two agents that are controlling each others functionality. A drawback of his approach is that as soon as one agent has been corrupted, the system must halt. Since a corrupted agent could accuse the other of being corrupted, it should not be possible for an agent to restart the other one. So, the joint task can only be successfully finished if no agent gets corrupted on its journey. This might be the reason that less attention has been payed to this idea.

We develop a model, in which we assume the agents not to communicate with their originator before they have finished their job. Otherwise the originator would be forced to stay online while his agents are working. The advantage of our proposal is, that it is based on a mathematical model for secure multi-party computation. Several such models have been published in the last decade. We have decided to use Canetti's recently proposed security model (see [Can01]) because it is tailored to represent communication networks like the Internet. Consequently, we do not have to demand all visited servers to be trustful. It is sufficient if the majority of our agents is not corrupted. This is achieved by distributing the computational state and all sensitive data redundantly over the participating agents.

The following presentation is structured into 3 sections. Section 2 is devoted to a brief survey of Canetti's model and its use for the agent setting. In the next section, we introduce a basic model in which the security problems arising by migration are still unsolved. The extended model in section 4 fills that gap.

2 Canetti's model and its implications

2.1 Canetti's model

Consider one community of n agents that has been created for a particular task T. By using a suitable migration control, there will be time periods in which no migration takes place and all of the agents are hosted by different servers. This setting is the same as that in secure multi-party computations because in that case several fixed servers participate in a joint calculation without requesting the servers to trust each other.

As mentioned above, our work is based on Canetti's definition of protocol security. Now, we are going to establish a basis for the presentation of our ideas in sections 3 and 4 by sketching Canetti's model. Main criteria for our decision for it's use was, that it provides security guarantees for arbitrary (even a priori unknown) concurrent environments with an asynchronous communication network, that delivers messages publicly, unauthenticated and without guaranteed message delivery.

Assume n servers jointly computing a functionality $\mathcal{F}$ which is realized by an n-party-protocol π. These servers are capable to participate concurrently in several protocol runs. Each of those executed programs is denoted as party. Furthermore, there is an adversary $\mathcal{A}$ in Canetti's model that is able to corrupt a limited number k of servers. In this case, it can read the entire state (including its history) and control the behaviour of these parties. Additionally, $\mathcal{A}$ has the power to read, modify, delay, and even delete outgoing messages of all n parties. Each entity is modeled as a Turing machine with two pairs of communication tapes. One for incoming/outgoing messages of the parties, the other one for local protocol input/output. Another adversarial entity $\mathcal{Z}$, which is called the *environment*, represents everything outside the current protocol execution. $\mathcal{Z}$ is responsible of delivering inputs to the parties since their origin is considered as external. Notice that, both adversarial entities are distinguishable by their knowledge and control. $\mathcal{A}$ knows and controls everything concerning messages between the parties, but is unaware of the inputs/outputs of the protocol, and for $\mathcal{Z}$ it is vice versa. Both are allowed to communicate with each other freely. The model is called "real-life-model". It is illustrated in figure 6 in appendix A.

For the definition of a secure protocol, one has to suppose an ideal setting. Obviously, no protocol execution can achieve more reliability than a protocol using a trusted entity which gets the inputs from all parties and returns (correct) outputs. A real-life-model supplemented with an unbounded number of such trusted entities for computing any functionality $\mathcal{F}$, is called $\mathcal{F}$-hybrid-model. A protocol π in the real-life-model is called secure, if

1. for any adversary attacking π there is one adversary in the $\mathcal{F}$-hybrid-model and
2. no possible environment is able to decide whether it acts in a protocol execution within the $\mathcal{F}$-hybrid or the real-life model.

Therefore, the most interesting cases are those in which the "interactive distinguisher" $\mathcal{Z}$ holds back some knowledge from $\mathcal{A}$. This enables $\mathcal{Z}$ to check whether the protocol outputs are correlated to this secret knowledge and, thus, might be able to differentiate the models.

Since our agent communities are supposed to work in the internet, we cannot presume the existence of a broadcast channel. On account of this, we need Byzantine Agreements (see [Gol95]) which limit the number k of corrupted parties to $n/3$ to obtain a protocol that is secure in the sense of Canetti's definition. For our agent setting, this implies that more than $2n/3$ of the hosts must be honest during each time interval in which no migration takes place.

2.2 Distributed computations

Since we want to secure the execution of arbitrary functions, we have to translate them into a k-robust protocol. This has to be done because an adversary in our model is limited to influence less than k inputs. Several protocol compilers have been developed. See for example [GMW87], [BGW88] and [CCD88]. In [GMW87] the resulting protocol is divided in two steps. At first, each party commits to its local input. To be able to detect a party that deviates from the protocol the other parties possess shares of everyone's randomness. The second part, the execution, is organised in several rounds. In each of them, every party is activated at least once to perform computations and to send messages. The correctness (in the sense of the protocol) of one party's activities are checked by the others through a zero-knowledge proof. Messages of one round must have been delivered until the beginning of the next round.

Canetti states in [Can01] that he does not know if [GMW87] is secure in his model. He proposes the use of [BGW88] which provides an information-theoretic secure synchronous protocol that stays secure in his setting. But also asynchronous networks can be handled by using the techniques of [BCG93] and [BKR94]. In [BGW88], the authors use a verifiable secret sharing scheme (VSS) to enable the community to detect improper or missing commitments in the first step. The actual evaluation of the function is done in the second phase.

2.3 Canetti Slices

To translate Canetti's model into a model for secure computations in multi-agent systems, we first have to fix all participating entities of the system.

Instead of commissioning one agent to fulfil a particular task, we use a community of agents, which share their global state of computation redundantly and solve the task in co-operation. For this purpose, the agents are able to communicate freely and to execute distributed computations. We consider every agent as one of Canetti's parties and every host as one of the servers (which are able to host several agents at the same time). There is only one adversary in Canetti's model. In multi-agent systems, every host has to be considered as possibly hostile. To manage this, we consider the community of malicious hosts controlled by a kind of "super-adversary". This is plausible because:

- In the worst case all malicious hosts co-operate and can be seen as one adversary.
- Any set of separately working adversaries cannot cause more damage to the entity of all n agents than one "super-adversary".

The "super-adversary" is consistent with Canetti's adversary and it is even stronger than any adversary that could exist in a real agent system.

Obviously, we maintain every security guarantee given by Canetti, as long as we only consider a time period in which no migration takes place. We call such a period *Canetti Slice*. During this time interval an agent community consisting of n agents is executed by n different hosts. What happens when a migration takes place? There again, we have n agents executed by n different hosts, but one of them is new.

3 A model for a secure mobile agent community

In this section we start by defining a basic agent and a basic protocol that demonstrates how a community with a distributed computational state could be realized. In this protocol we include a very rudimentary migration process. Any functionality that could be used to solve a user-defined task T can be realised by such a protocol. Several security risks arising by migration are not handled here, but will be treated in the next section.

3.1 The basic agent

Let A_j be one of the n mobile agents, which have been designed for the fulfilment of a task T. Like the classical agent, our basic agent can be roughly divided into code and data. Its code C is the same as that of the other agents of his community, but it would also be possible to provide it with an unique code. In any case C contains the information about the size n of the agent community.

The agent's data consists of shared knowledge. Therefore, it is confidential as long as an adversary has not enough shares to reconstruct the secret information. Unfortunately, in the basic model, the adversary is able to collect enough shares over the time. Later on we solve this problem by resharing methods.

During its travel, A_j enters a series of hosts $H_0, H_{j1}, \dots, H_{jm}$, whereby H_0 is the one, on which he has been initialised. Entering a host H_{ji}, the agent's database consists of a set s_j of shares that have been added as a result of distributed computations by one of its preceeding hosts. Parts of s_j are shares of a list Q that is used to control the migration process and the entire knowledge about a location list L_c. Unnecessary or redundant knowledge may be deleted. This implies, that it is not always possible to detect the supplier of wrong knowledge after the completion of the task.

3.2 The basic protocol

The protocol is divided into an initialisation phase on a trusted host H_0 and the execution/migration phase. Any communication between hosts is assumed to be done through a secure channel. Every message contains a community id, which enables the receiver to assign it to one of the agents hosted by him. Messages originated by an agent that is not a member of the community are ignored. In the protocol this can be checked by a location list L_c.

In the following, we present the necessary subroutines that have to be executed by a host on demand of the protocol:

The subroutine `deliver`
The function `deliver` has 2 parameters: a list $L' \subseteq L_c$ of receivers and a message m. If A_k is the first element of Q[1] and the current host of A_k is in L', then the message m is buffered. The message m is sent to all confirmed members of L'.

The subroutine `run`
`run` is the most important function in our model. It is used to invocate k-robust n-party sub-protocols, which are executed by the community. The function's parameters are: the current location list L_c, a protocol X, and an input r for the protocol X. The input r is given to the local program that is part of the new protocol instance of X. It contains randomness and possibly additional information.

The next host is determined by execution of the sub-protocol `migrate` (see figure 1). The protocol could be invoked concurrently. Therefore, we require the termination of the current migration process before the next one is going to be processed. The first element of Q can be used to check which call of `migrate` belongs to the current migration process.

[1] In this case, A_k is migrating, but his next host is not yet confirmed.

The functionality of sub-protocol `migrate`

1. If the input contains "Q" then
   If not more than $2n/3$ of "$Q+$" or "$Q-$" of such calls arrived
   store this request and exit.
   Else
   If #"$Q+$"$> 2n/3$ then inform every host to update L_c and to send all messages that have been buffered for the first element in Q to the new host.
   Remove the first element of Q.
   Send a termination message to the old host.
   If $Q \neq \{\}$ then continue the protocol for the first element of Q.
   Else exit.
   Else
   append the request to Q
   If $|Q| \geq 1$ exit.
2. Distributed computation of A_j's next host $H_{j(i+1)}$. It is not allowed to choose a member of L_c.
3. Broadcast of all resulting shares to all current hosts.
4. Reconstruction of $H_{j(i+1)}$ by each host.
5. Each host sends its L_c to $H_{j(i+1)}$.

Fig. 1. Functionality of `migrate`

Initialisation

1. The originator divides a database D in n redundant shares and distributes them among the agents. Furthermore, each agent is provided with a code C.
2. H_0 computes a list $L_c = [H_{11}, \ldots, H_{n1}]$ containing the hosts of the first Canetti Slice.
3. For all $1 \leq j \leq n$, the host H_0 sends the message (A_j,"Agree?",H_0) to Host H_{j1}.
4. While there is any j with outstanding positive response:
   If H_{j1} sends "no", H_0 determines a new H_{j1}, updates L_c and sends (A_j,"Agree?",H_0) to H_{j1}.
   If H_{j1} sends "yes", H_0 makes an endorsement about H_{j1}.
5. H_0 sends L_c to all members of L_c.

Migration cycle of A_j on host H_{ji} ($i \geq 1$)

1. H_{ji} makes a decision *dec* $\in$ {"yes","no"} about the execution of A_j.
2. If $H_{j(i-1)} = H_0$ then `deliver`(*dec*, H_0),
   else
   while not more than $2n/3$ location lists L_c are available, store incoming location lists sent by different servers.
   Fix the current location list L_c of all agents by a majority decision.
   Then, H_{ji} executes `deliver`(*dec*, L_c).
   If *dec* = "no", H_{ji} deletes the agent,
   else H_{ji} starts the execution of A_j.
3. During the execution the following events may occur
   – Calls of the function `run`((r,"$Q+$"), L_c,`migrate`) if any host sends a positive response concerning the execution of an agent. Calls of `run`((r,"$Q-$"), L_c,`migrate`) in case of a negative response.

- Local computations on shares
- Invocation of a subroutine $\texttt{run}(r, L, X)$ for distributed computations
- Updates of the set of shares s_j
- Forwarding of messages to A_j
- Delivery of messages by execution of $\texttt{deliver}(L', m)$
- A_j demands its migration, therefore, H_{ji} calls $\texttt{run}((r,"A_j"), L_c, \texttt{migrate})$
- Receipt of the next host $H_{j(i+1)}$. H_{ji} exits the event loop.

4. After receiving the next host $H_{j(i+1)}$, the agent A_j is sent to it with the plea for an agreement response.

   While there is no termination message from c:

   If there is a positive answer from $H_{j(i+1)}$, then H_{ji} calls $\texttt{run}((r,"Q+"), L_c, \texttt{migrate})$.

   If there is a negative answer from $H_{j(i+1)}$, then H_{ji} calls $\texttt{run}((r,"Q-"), L_c, \texttt{migrate})$.

   H_{ji} deletes A_j.

3.3 Discussion

We consider an agent as corrupted when it has been maliciously modified on a host. Additionally, its shares could be spied out by a host. But we do not consider such an agent as corrupted because we will use a suitable resharing method in our extended protocol to make them useless.

Like in the raw Canetti model, we assume a corrupted basic agent to stay in this condition for the rest of its life. Thus, the probability of having an agent community with less than $n/3$ corrupted members decreases over time.

In the basic protocol, we do not require any time constraints for the migration process or the execution on a host. Therefore, a malicious host can grind the whole community to a halt by refusing to send an agreement message after receiving an agent with the request to host it. The lack of a timeout allows a malicious host to retain the agent forever by never issuing a migration request. This does not change anything in the basic model because at that moment the agent is already corrupted.

The advantages of distributed computations are the guaranteed confidentiality of data and the correct execution of an user-defined functionality as long as less than $n/3$ of the agents are corrupted or spied out. This is a direct result given by [Can01]. For a hostile environment like the one autonomous agents are living in, this is already a quite strong guarantee. Nonetheless, for practical reasons we are going to handle the problems mentioned above by requiring the agents of one community to control each other and, if necessary to clean an agent that became corrupted. Additionally, we introduce a suitable resharing method that is performed regularly.

4 An extension

The previously discussed security risks mainly arise from the transition from one Canetti Slice to another. So, this section is dedicated to enrich our model with techniques to secure the transition by share renewal as well as detecting and cleaning of corrupted agents. Additionally, we introduce some features like authentication and local computations on public data. The latter enables the originator to decide whether a particular computation needs to be performed securely and with non-negligible communication complexity. Otherwise, they could be performed insecurely but locally and efficiently.

If code and data are digitally signed, any changes can be detected. Obviously, the code can be signed by the originator. The public data is always signed by the host that produced it. But who signs the private data? It should be signed by the community c because of the following two reasons:

- If an agent gets lost, the community is able to replace it by copying the code and reconstructing the private data.
- No host can join a distributed computation with correct shares and insert signed but faked shares into the agent's database without being detected later on.

We assume the existence of a public-key-infrastructure with certification authorities. So, everybody is able to get someone's public key in a reliable way. This implies, that every host can check if the code and/or data of the agent has been changed without permission.

4.1 The extended agent

Entering the ith host H_{ji}, the agent A_j consists of

- a list $K = [(p_0, s_0, O, c, H_0, t_m, t, \text{sig}_{H_0}(h(p_0)), \text{sig}_c(h(s_0|O|c|H_0|t_m|t))),$
  $(p_{jk}, s_{jk}, \text{sig}_{H_{jk}}(h(p_{jk})), \text{sig}_c(h(s_{jk})))|1 \leq k \leq i-1]$
- a signature $\text{sig}_O(h(C))$

whereby for all $0 \leq k \leq i-1$, p_{jk} is the public knowledge and s_{jk} is the set of shares added by host H_{jk} ($H_{j0} = H_0$). The set s_0 additionally contains some system information like shares of:

- c's private key
- counters c_{A_l} for each agent A_l (needed for the migration process of agent A_l)
- c_m that counts the number of migration trials in current migration process
- the list Q that is the queue for the concurrent migration requests
- the location list L_c
- n history lists L_{h_j}
- $Q[1]$

From the last three entries the agent possesses enough shares to be able to reconstruct the data for its own. The initial shares are concatenated with a number t_m that limits the number of migration trials within one migration process, the maximum execution time t on one host, the originator's name O, the community's identity c and the initial host H_0.

4.2 The extended protocol

In the extended protocol, the time available for the migration process and execution is controled because every host owns a timer t_{A_j} for every agent of c. When a new host is computed for an agent, every host resets the timer that has been assigned to the agent. After a successful migration all current timer values are submitted to the new host by his predecessor. As soon as a particular timer t_{A_j} runs out, the host calls the sub-protocol `count` to increase a counter c_{A_j}. For details about `count` see figure 2. When the counter

The functionality of sub-protocol `count`

1. Increase the distributed counter c_{A_l} by 1
2. If $c_{A_l} > 2n/3$, then
   increase c_m by 1
   If $c_m \leq t_m$, then
      execution of sub-protocol `ext_migrate`
   Else
      execution of sub-protocol `create`

Fig. 2. Functionality of `count`

exceeds $2n/3$ and not too many trials have failed, a migration process for agent A_j starts by calling the sub-protocol `ext_migrate` (see figure 3). Otherwise the agent is recreated. Before executing an agent, every host checks the correctness of all signatures. By this,

The functionality of sub-protocol `ext_migrate`

1. If the input contains "Q" then
   If not more than $2n/3$ of "$Q+$" or "$Q-$" calls arrived
      store this request and exit.
   Else
      If #"$Q+$"$> 2n/3$ then inform every host to update L_c and to send all messages
         that have been buffered for the first element in Q to the new host.
         Send a termination message to the old host.
         Call sub-protocol `reshare` and remove the first element of Q.
      Else $c_m := c_m + 1$;
         If $c_m > t_m$ then call sub-protocol `create` for $Q[1]$ and exit.
      If $Q \neq \{\}$ then continue the protocol for the first element of Q.
      Else exit.
   Else
      append the request to Q
      If $|Q| \geq 1$ then exit.

2.–5. analogously to steps 2–5 of figure 1

6. Every host sends $L_{h_j} = \{H_0, H_{j1}, \ldots, H_{ji}\}$ to $H_{j(i+1)}$ [a].
7. Every host starts a timer t_{A_j}.

[a] In case of the renewal of an agent, the hosts additionally send their shares of the new agent to the new location.

Fig. 3. Functionality of `ext_migrate`

the data/code integrity is verified, too. To enable the host to retrieve the relevant public keys the history list L_{h_j} is submitted to the new host by all other hosts. The history lists of the other agents are submitted, too. This guarantees the integrity of those lists. A distributed storage is also possible and more efficient, but for sake of simplification not used here. If the integrity of any of the private data or the code is violated, the host calls for a sub-protocol `create` (see figure 4) that prompts the community to distributedly compute a new agent and to send it to a new host. This agent has no initial public data.

The functionality of sub-protocol `create`

1. If the request contains "A_k", then check if the sender of the request is the current host of A_k. If not, then exit.
2. Fix a code C and the signature $\text{sig}_O(h(C))$ by a majority decision
3. Computation of new shares $s_0, \ldots, s_i$ [a]
4. Fixing the history L_{h_j} by a majority decision and append the separator $\diamond$.
5. Execution of `ext_migrate`

[a] This can be done analogously to [OY91]

Fig. 4. Functionality of `create`

After receiving a positive agreement response concerning the execution of an agent from a new host, a host calls `run`$((r,"Q+"), L,$ `ext_migrate`$)$. In case of a negative response, `run` is called with "$Q-$". The protocol is executed as soon as more than $2n/3$ calls for "$Q+$" resp. "$Q-$" from different servers arrived. It instructs the hosts to update their L_c, to send the buffered messages to the new host and to renew the shares by invocating the sub-protocol `reshare`. If the agent's shares never expire, the "super-adversary" might be able to collect enough of them to gain full information about the community's secret. Therefore, it is inevitable to renew the shares. We propose the use of the technique of [OY91] because it is based on the secure distributed computation scheme in [BGW88] which we used before. The method makes use of the verifiable secret sharing scheme which is based on Shamir's secret sharing algorithm presented in [Sha79]. If the "super-adversary's" abilities are more restricted, one could delay the resharing process until the first agent wants to migrate for the x^{th} time. The number x depends on the assessment of the network and can be counted by the lists L_{h_k}.

The functionality of sub-protocol `reshare`

1. Distributed computation of new share sets s_{kl} and s_0 for each k, l as per [OY91]
2. Distributed signature of the new sets with the private key of the community
3. Each host deletes the old shares.

Fig. 5. Functionality of `reshare`

Initialisation

1. The originator divides D in n shares and distributes them among the agents. Furthermore every agent gets some public data p_0 and a code C.
2. H_0 computes a list $L_c = [H_{11}, \ldots, H_{n1}]$ containing the hosts of the first Canetti Slice.
3. For all $1 \leq j \leq n$, the host H_0 sends the message (A_j,"Agree?",H_0) to Host H_{j1} and starts timers t_{A_j}.
4. As long as there is any j with outstanding positive response:

   If any timer t_{A_j} runs out or H_{j1} sends "no", then
   compute a new host H_{j1} for A_j, send (A_j,"Agree?",H_0) to H_{j1} and
   restart timer t_{A_j}.
   else
   If H_{j1} sends "yes" then t_{A_j} is stopped and H_0 makes an endorsement about j.

5. H_0 sends L_c and for every $1 \leq j \leq n$, a list $L_{h_j} = \{H_0\}$ containing the previous hosts of agent A_j to all members of L_c. Additionally, each host gets timers $t_{A_1} = \ldots = t_{A_n} = 0$.

Migration cycle of A_j on host H_{ji} ($i \geq 1$)

1. H_{ji} checks $\mathrm{sig}_O(h(C))$.
2. H_{ji} makes a decision $dec \in \{$"yes","no"$\}$ about the execution of A_j.
   If $dec =$ "yes", all timers t_{A_k} are continued.
3. If $H_{j(i-1)} = H_0$, then `deliver`(dec, H_0).
   Else
   Incoming location lists L_c and history lists L_{h_k} sent by any server are stored.
   If more than $2n/3$ lists L_c are available, a majority decision is made to fix L_c.
   If L_c exists, H_{ji} executes `deliver`(dec, L_c).

   If $dec =$ "yes",
   while for at least one k the list L_{h_k} is not yet fixed do
   as soon as for one k, $1 \leq k \leq n$, more than $2n/3$ lists L_{h_k} are available
   L_{h_k} is fixed by a majority decision.

   Else H_{ji} deletes the agent.
4. H_{ji} checks for $1 \leq k \leq i-1$ the signatures of s_{jk} by means of L_{h_j}. The public knowledge $p_{jk'}$ is checked by all entries of L_{h_j} after the last separator $\diamond$.
   If the check of any s_{jk} or the code fails then call `run`((r,"A_j"), L_c, `create`) and delete A_j.
   else start the execution of A_j.
5. During the execution, the following events may occur
   - Calls of the function `run`((r,"$Q+$"), L_c, `ext_migrate`) in case of a positive agreement response from any host. Call of `run`((r,"$Q-$"), L_c, `ext_migrate`) in case of a negative response.
   - Local computations on shares

- Invocation of a subroutine $\texttt{run}(r, L_c, X)$ for distributed computations
- Updates of the set of shares s_{ji}
- Forwarding of messages to A_j
- Delivery of messages by execution of $\texttt{deliver}(L', m)$
- A timer t_{A_k} runs out. Then call $\texttt{run}((r,"A_k"), L_c, \texttt{count})$.
- A_j demands its migration, therefore, H_{ji} calls $\texttt{run}((r,"A_j"), L_c, \texttt{ext\_migrate})$
- Receipt of the next host $H_{j(i+1)}$. H_{ji} exits the event loop.

6. After receiving the next host $H_{j(i+1)}$, the agent A_j is sent to it with the plea for an agreement response.

   While there is no termination response from c:
   If there is a positive answer from $H_{j(i+1)}$, then H_{ji} calls $\texttt{run}((r,"Q+"), L_c, \texttt{ext\_migrate})$.
   If there is a negative answer from $H_{j(i+1)}$, then H_{ji} calls $\texttt{run}((r,"Q-"), L_c, \texttt{ext\_migrate})$.
   If a timer t_{A_k} runs out, H_{ji} calls $\texttt{run}((r,"A_k"), L_c, \texttt{count})$.

   H_{ji} deletes A_j and stops all timers. The current values of all timers are transmitted to $H_{j(i+1)}$.

4.3 Discussion

Our novel approach improves the security of mobile computations significantly. In particular, the following security features are achieved:

- Authentication of the agents.
- Sensible data can be kept confidential since they are stored distributedly and any computation on them is done distributed, too.
- Violation of code/data integrity can be detected. Code and private data can be restored.
- For every sensible computation we use $n/3$-robust protocols which provide us with guarantees for their private and correct execution.
- Using timeouts, we guarantee that a malicious host is not able to flood the community with useless requests for a long time. An agent cannot be held forever.
- Malicious routing is impossible as long as less than 1/3 of the agents are corrupted at the same time.
- The community is self-repairing. Therefore, in a real system the preconditions for Canetti's security model could be met at every time.

To the best of our knowledge, this is the first model that achieves security for mobile computations. We are convinced of its importance for future design of multi-agent systems even though its communication complexity is high.

References

[BCG93] Michael Ben-Or, Ran Canetti and Oded Goldreich, "Asynchronous Secure Computations", *25th Symposium on Theory of Computing (STOC)*, pp.52–61, ACM 1993.

[BGI01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan and Ke Yang, "On the (Im)possibility of Obfuscating Programs", *Advances in Cryptology—Crypto 2001*, Lecture Notes in Computer Science, Vol. 2139, pp. 1–18, Springer Verlag, 2001.

[BGW88] Michael Ben-Or, Shafi Goldwasser and Avi Wigderson, "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation", *20th Symposium on Theory of Computing (STOC)*, ACM, pp. 1–10, 1988.

[BKR94] Michael Ben-Or, Boaz Kelmer and Tall Rabin, "Asynchronous Secure Computations with Optimal Resilience", *13th PODC*, pp. 183–192, 1994.

[Can01] Ran Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols", *Cryptology ePrint Archive*, Report 2000/067.

[CCD88] David Chaum, Claude Crépeau and Ivan Damgard, "MULTYPARTY UNCONDITIONALLY SECURE PROTOCOLS", *Proceedings of the 20th Annual Symp. on the Theory of Computing*, pp. 11–19, ACM, 1988.

[GMW87] Oded Goldreich, Silvio Micali and Avi Wigderson, "How to Play any Mental Game", *Proceedings of STOC 87*, pp. 218–229, 1987.

[Gol95] Oded Goldreich, "Foundations of Cryptography (Fragments of a book)", Weizmann Institute of Science, 1995, available at `http://theory.lcs.mit.edu/~oded/frag.html`.

[Hoh97] Fritz Hohl, "An Approach to Solve the Problem of Malicious Hosts", *Fakultätsbericht*, Universität Stuttgart, Fakultät für Informatik, Germany, Vol. 1997/03, 1997.

[LM99] Sergio Loureiro and Refik Molva, "Function Hiding based on Error Correcting Codes", *Proceedings of Cryptec '99 – International Workshop on Cryptographic Techniques and Electronic Commerce*, pp. 92–98, July 1999.

[Rot99] Volker Roth, "Mutual protection of co-operating agents", In Jan Vitek, and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, Lecture Notes in Computer Science, Vol. 1603, pp. 2bb75–285, Springer Verlag, 1999.

[Sha79] Adi Shamir, "How to Share a Secret", *Communications of the ACM*, Vol. 24, No. 11, pp. 612–613, 1979.

[ST98] Tomas Sander and Christian F. Tschudin, "Towards Mobile Cryptography", *Proceedings of the IEEE Symposium on Security and Privacy*, 1998.

[OY91] Rafail Ostrovsky and Moti Yung, "How To Withstand Mobile Virus Attacks", *Proc. of the 10th Ann. ACM Symp. on Principles of Distributed Computing*, pp. 51–59, ACM SIGACT and ACM SIGOPS, 1991.

[Wei99] Gerhard Weiss, "Multiagent Systems", edited by Gerhard Weiss, MIT Press, 1999.

A Illustration of Canetti's model

The following figure 6 illustrates the concurrent execution of three protocols π_1, π_2 and π_3 on N servers. Thereby, π_i is an n_i-party protocol ($1 \leq i \leq 3$). The adversary $\mathcal{A}$ of a protocol π_i is able to read and write on the communication tapes of each participiciant of this protocol. In the figure, this is represented by the arrows.

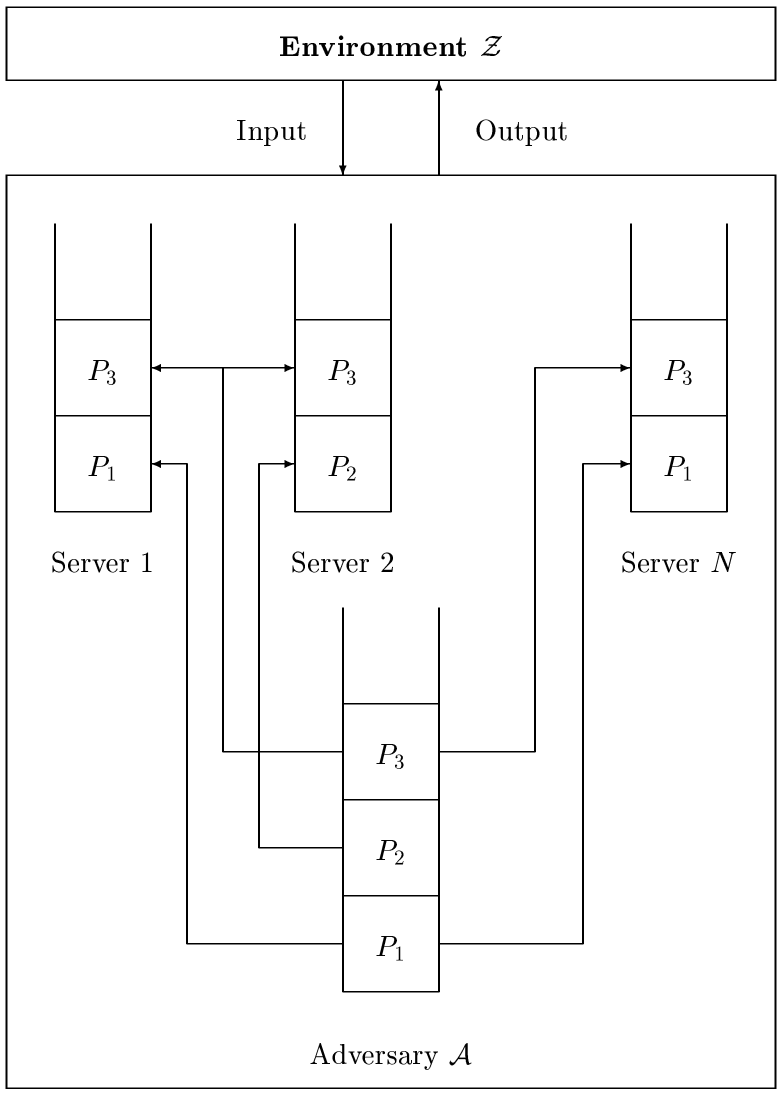

Fig. 6. The real-life-model with multi-party protocols π_1, π_2 and π_3