Process calculi for the verification of security properties of communication protocols for Peer-to-Peer systems

Hugo Andrés López Andrés Aristizábal Camilo Rueda[*]
Frank D. Valencia[†]

February 6, 2006

Research Report [1]

Abstract

Recent advances in communication have made the use of dynamic and reconfigurable network topologies a mandatory requirement in scenarios where the participants must actively collaborate to each other to achieve a common, specific goal. A particular case of those scenarios are the Peer-to-Peer (P2P) systems. The wide applicability of P2P-based applications and its pervasive presence in corporate applications are two important factors that suggest a careful study of the communication protocols underlying these systems. Surprisingly, little effort has been invested in giving formal foundations that support both protocols and its security properties. This paper is intended to be a step towards such a research strand, using a process calculus as main resource to build such foundations. In particular, we focus on the reconfiguration problem in P2P-based applications. We propose two protocols for modeling this problem: while the first one is inspired on an existing protocol (known as the Friends Troubleshooting Network or FTN), the second constitutes an original proposal based on a multi-layered encryption system. We show how SPL (the process calculi in which both models are given) is well-suited to model and to proof certain security properties of the protocols.

1 Introduction

Collaborative P2P applications aim to allow application-level collaboration between users. The inherently ad-hoc nature of P2P technology makes it a good fit for user-level collaborative applications. These applications range from instant messaging and chat, to online games, to shared applications

[*]Pontificia Universidad Javeriana - Cali
[†]LIX Ecole Polythecnique
[1]This report was prepared in the context of the research project "Modelamiento de Problemas en Ciencia y Tecnología Usando Cálculos de Procesos Concurrentes", founded by Pontificia Universidad Javeriana - Cali

[Ese02, BS04, GK03, BMWZ05, Rip01] that can be used in business, educational, and home environments. Unfortunately, a number of technical challenges remain to be solved before pure P2P collaborative implementations become viable, such as location discovery, fault tolerance, network constraints and security [MKL$^+$02]. Concerning to the security of the system, P2P systems are used to share private information between peers over open networks, involving properties like secrecy, anonymity and non-traceability which have been studied in the literature in order to overcome such risks [MKL$^+$02].

SPL is a formalism that allows to model and analyse security protocols using an open network approach. The power of this process calculus lies in its formal specification, and the use of reasoning techniques as a way to prove security properties in a concurrent communication system. To the best of our knowledge, only authentication protocols are modelled and formalised in this process language [CCM02, Cra03, Mil02]. Therefore, the study of other kinds of protocols, possibly involving different security properties to be verified, is very important nowadays.

In this paper we explore how can SPL reasoning techniques can serve as well for the analysis of a collaborative P2P system. We use a cutting-edge system as a valid case of study to achieve this affirmation. This system is intended to resolve the problem of automatic reconfiguration of applications in a fully distributed system without compromising the identities of the agents involved in the protocol, neither their own secrets.

We follow a two-fold approach for tackling the problem of dynamic reconfiguration of applications in P2P systems. Firstly, we extend the basic syntactic structure of SPL with some notions of concurrency relevant to security, to formalise an SPL model for the Friends Troubleshooting Network (FTN) protocol [WHY$^+$04a]. Secondly, we propose a new protocol that maintains the main functionality of the FTN protocol, in a model much concise and less complex than the proposed by Wang et al. In order to do so, we heavily use the idea of a layered encryption protocol [GRS99].

The document is structured as follows. In the following section we introduce some basic notions of SPL [Cra03]. In section 3 we explain the problem of dynamic reconfiguration of P2P-based applications, taking the FTN network architecture as base. A definition of the essential properties to be ensured in this kind of systems is given in section 4. In section 5 we extend the basic syntax of SPL by means of a set of encodings, to enable a formal model for the FTN protocol. Then in section 6 we give a formalisation for a new and more concise protocol with the same functionality as FTN. The DR protocol is verified using the basic proof structure inherent to SPL. In the las section we give a brief discussion about the importance of our contributions.

2 Preliminaries

This section presents a brief overview of SPL (Security Protocol language), an economic secure process calculus close to an asynchronous Pi-Calculus proposed by Winskel and Crazzolara et. a [CW01], Conceptual design and the full coverage of the calculus are deeply explained in [Cra03].

2.1 SPL

SPL is a process calculus designed to model protocols and prove their security properties by means of transitions and event-based semantics. SPL is based on the Dolev-Yao Model. The calculus is operationally defined in terms of configurations containing items of information (messages) which can only increase during evolution, modelling the fact that in an open network an intruder can see and remember any message that was ever in transit.

2.1.1 SPL Syntax

The syntactic entities SPL are described below:

- An infinite set N of names denoted by $n, m, ..., A, B, ...$ Names range over *nonces* (randomly generated values, unique from previous choices [Per96]) and agent names.
- Three types of variables: over names (denoted by $x, y, ..., X, Y, ...,$), over keys ($\chi, \chi', \chi_1, ...,$) and over messages ($\psi, \psi', \psi_1, ...,$). They could also be expressed as a vector of variables, denoted as $\vec{x}\vec{\chi}\vec{\psi}$ respectively.
- A set of process, denoted by $P, Q, R, ...$

Variables over names	$x, y, ..., X, Y, ...,$
Variables over keys	$\chi, \chi', \chi_1, ...,$
Variables over messages	ψ, ψ', ψ_1
Name expressions	$v \ ::= \ n, A, ... \mid x, X$
Key expressions	$k \ ::= \ Pub(v) \mid Priv(v) \mid Key(\vec{v}) \mid \chi, \chi', ...$
Messages	$M, M' \ ::= \ v \mid k \mid (M, M') \mid \{M\}_k \mid \psi, \psi', ...$
Processes	$p \ ::= \ out\, new(\vec{x})\, M.p \mid in\, pat \vec{x}\vec{\chi}\vec{\psi}\, M.p \mid \ \Vert_{i \in I}\, P_i \mid !P$

Table 1: SPL Syntax

Output	$\langle out\, new(\vec{x})M.p, s, t\rangle \overset{outnew(\vec{n})M[\vec{n}/\vec{x}]}{\longrightarrow} \langle p[\vec{n}/\vec{x}], s \cup \{\vec{n}\}, t \cup \{M[\vec{n}/\vec{x}]\}\rangle$	if all the names in $\vec{n}$ are distinct and not in s
Input	$\langle in\, pat\, \vec{x}\vec{\chi}\vec{\psi}M.p, s, t\rangle \overset{inM[\vec{n}/\vec{x}, \vec{k}/\vec{\chi}, \vec{N}/\vec{\psi}]}{\longrightarrow} \langle p[\vec{n}/\vec{x}, \vec{k}/\vec{\chi}, \vec{N}/\vec{\psi}], s, t\rangle$	if $M[\vec{n}/\vec{x}, \vec{k}/\vec{\chi}, \vec{N}/\vec{\psi}] \in s$
Par. Comp.	$\dfrac{\langle p_j, s, t\rangle \xrightarrow{\alpha} \langle p'_j, s', t'\rangle}{\langle \Vert_{i \in I} P_i, s, t\rangle \xrightarrow{j:\alpha} \langle \Vert_{i \in I} P'_i, s', t'\rangle}$	where $p'_i = p'_j$ for $i = j,$ otherwise $p'_i = p_i$

Table 2: SPL Transition Semantics

The rest of the elements of SPL syntactic set are defined in Table 1, where $Pub(v)$, $Priv(v)$ and $Key(\vec{v})$ denote the generation of public, private and shared keys respectively. We use the vector notation $\vec{s}$ to denote a list of elements, possibly empty, $s_1, s_2, \ldots, s_n$.

2.1.2 Intuitive Description and Conventions

Let us now give some intuition and conventions for SPL processes.

The output process $out\, new(\vec{x})\, M.p$ generates a set of fresh distinct names (nonces) $\vec{n} = n_1, n_2, \ldots, n_m$ for the variables $\vec{x} = x_1, x_2 \ldots x_m$. Then it outputs the message $M[\vec{n}/\vec{x}]$ (i.e., M with each x_i replaced with n_i) in the store and resumes as the process $p[\vec{n}/\vec{x}]$. The output process binds the occurrence of the variables $\vec{x}$ in M and p. As an example of a typical output, $p_A = out\, new(\vec{x})\, \{x, A\}_{Pub(B)}.p$ can be viewed as an agent A posting a message with a nonce n and its own identifier A encrypted with the public key of an agent B. We shall write $out\, new(\vec{x})\, M.p$ simply as $out\, M.p$ if the vector $\vec{x}$ is empty.

The input process $in\, pat\, \vec{x}\vec{\chi}\vec{\psi}\, M.p$ is the other binder in SPL binding the occurrences of $\vec{x}\vec{\chi}\vec{\psi}$ in M executing p. As an example of a typical input, $p_B = in\, pat\, x, Z\, \{x, Z\}_{Pub(B)}.p$ can be seen as an agent B waiting for a message of the form $\{x, Z\}$ encrypted with its public key B: If the message of p_A above is in the store, the chosen instantiation for matching the pattern could be the alpha conversion $\{n/x, A/Z\}$, where n matches x and A does the same with Z. When no confusion arises we will sometimes abbreviate $in\, pat\, \vec{x}\vec{\chi}\vec{\psi}\, M.p$ as $in\, M.p$.

Finally, $\|_{i\in I}\ P_i$ denotes the parallel composition of all P_i. For example in $\|_{i\in\{A,B\}}\ P_i$ the processes P_A and P_B above run in parallel so they can communicate. We shall use $!P = \|_{i\in\omega}\ P$ to denote an infinite number of copies of P in parallel. We sometimes write $\|_{i\in\{1,2,\ldots n\}}\ P_i$. to mean $P_1 \parallel P_2 \parallel \ldots \parallel P_n$

The syntactic notions of free variables and closed process/message are defined in an usual way. A variable is *free* in a process/message is has a non-bound occurrence in that process/message. A process/message is said to be *closed* if it has no free variables.

2.1.3 Transition Semantics

SPL has a transition semantics over configurations that represents the evolution of processes. A configuration is defined as $\langle p, s, t\rangle$ where p is a closed process term (the process currently executing), s a subset of names $\mathbf{N}$ (the set of nonces generated so far), and t is a subset of variable-free messages (i.e., the store of output messages).

The transitions between configurations are labelled by *actions* which can be input/output and maybe tagged with an index i indicating the parallel component performing the action. Actions are thus given by the syntax $\alpha ::= out\, new(\vec{n})\, M \mid in\, M \mid i : \alpha$. where $\vec{n}$ is as a set of names, i as an index and M a closed message.

Intuitively a transition $\langle p, s, t\rangle \xrightarrow{\alpha} \langle p', s', t'\rangle$ says that by executing α the process p with s and t evolves into p' with s' and t'. The new set of messages t' contains those in t since output messages are meant to be read but not removed by the input processes. The rules in Table 2 define the transitions between configurations. The rules are easily seen to realize the intuitive behaviour of processes given in the previous section.

Nevertheless, SPL also provides an *event based semantics*, where events of the protocol and their dependencies are made more explicit. This is advantageous because events and their pre and post-conditions form a Petri-net, so-called SPL nets.

2.1.4 Event-Based Semantics

Although transition semantics provide an appropriate method to show the behaviour of configurations, these are not enough to show dependencies between events, or to support typical proof techniques based on maintenance of invariants along the trace of the protocols. To do so, SPL presents an additional semantics based in events that allow to explicit protocol events and their dependencies in a concrete way.

SPL event-based semantics are strictly related to persistent Petri nets, so called *SPL-nets* [Cra03] defining events in the way they affect conditions. The reader may find full details about Petri Nets and all the elements of a SPL-Nets in [Cra03], below we just recall some basic notions.

Description of Events in SPL In the event-based semantics of SPL, conditions take an important place as they represent some form of local state. There are three kinds of conditions: *control*, *output* and *name* conditions (denoted by C, O and N, respectively). C-conditions includes input and output processes, possibly tagged by an index. O-conditions are the only persistent conditions in SPL-nets and consists of closed messages output on network. Finally, N-conditions denotes basically the set of names $\mathbf{N}$ being used for a transition. In order to denote pre and post conditions between events, let ${}^{\cdot}e = \{{}^{c}e, {}^{o}e, {}^{n}e\}$ denote the set of control, name and output preconditions, and $e^{\cdot} = \{e^{c}, e^{o}, e^{n}\}$ the equivalent set of postconditions. An SPL event e is a tuple $e = ({}^{\cdot}e, e^{\cdot})$ of the preconditions and postconditions of e and each event e is associated with a unique action $act(e)$. Figure 1 gives the

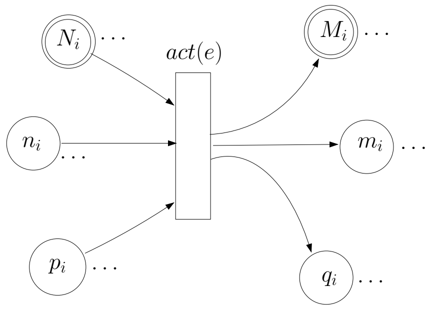


Figure 1: Events and transitions of SPL event based semantics: p_i and q_i denote control conditions, n_i and m_i name conditions and N_i, M_i output conditions. Double circled conditions denote persistent events.

general form of an SPL event. The exact definition of each element of the events can be found in [Cra03].

To illustrate the elements of the event semantics, consider a simple output event $e = (\mathbf{Out}(out\ new\vec{x}M); \vec{n})$, where $\vec{n} = n_1 \dots n_t$ are distinct names to match with the variables $\vec{x} = x_1 \dots x_t$. The action $act(e)$ corresponding to this event is the output action $out\,new\vec{n}M[\vec{n}/\vec{x}]$. Conditions related with this event are:

$$\begin{array}{lll} {}^{c}e = \langle out\,new(\vec{x}).M.p, a\rangle & {}^{o}e = \emptyset & {}^{n}e = \emptyset \\ e^{c} = \langle Ic(p[\vec{n}/\vec{x}])\rangle & e^{o} = \{M[\vec{n}/\vec{x}]\} & e^{n} = \{n_1, \dots n_t\} \end{array}$$

Where $Ic(p)$ stands for the initial control conditions of a closed process p: The set $Ic(p)$ is defined inductively as $Ic(X) = \{X\}$is X is an input or an output process, otherwise $Ic(\|_{i\in I}\ P_i) = \bigcup_{i\in I}\{i : c \mid c \in Ic(P_i)\}$

2.1.5 Relating Transition and Event Based Semantics

Transition and event based semantics are strongly related in SPL by the following theorem from [Cra03].

Theorem 1. *i)* If $\langle p, s, t\rangle \xrightarrow{\alpha} \langle p', s', t'\rangle$, then for some event e with $act(e) = \alpha$, $Ic(p) \cup s \cup t \xrightarrow{e} Ic(p') \cup s' \cup t'$ *in the SPL-net.*

ii) If $Ic(p) \cup s \cup t \xrightarrow{e}$ M' in the **SPL**-net, then for some closed process term p', for some $s' \subseteq N$ and $t' \in O$, $\langle p, s, t\rangle \xrightarrow{act(e)} \langle p', s', t'\rangle$ and M' $= Ic(p') \cup s' \cup t'$.

Justified in the theorem above, the following notation will be used: Let e be an event, p be a closed process, $s \subseteq N$, and $t \subseteq O$. We write $\langle p, s, t\rangle \xrightarrow{e} \langle p', s', t'\rangle$ iff $Ic(p) \cup s \cup t \xrightarrow{e} Ic(p') \cup s' \cup t'$ in the **SPL**-net.

2.1.6 Events of a Process

Each process has its own related events, and for a particular closed process term p, the set of its related events $Ev(p)$ is defined by induction on size, in the following way:

$Ev(out\,new\,\vec{x}M.p) \quad = \quad \{$ **Out** $(out\,new\,\vec{x}M.p;\ \vec{n})\} \cup \bigcup\{Ev(p[\vec{n}/\vec{x}])\}$
Where $\vec{n}$ are distinct names
$Ev(in\,pat\,\vec{x}\vec{\chi}\vec{\psi}M.p) \quad = \quad \{$**In**$(in\,pat\,\vec{x}\vec{\chi}\vec{\psi}M.p;\ \vec{n}, \vec{k}, \vec{L})\} \cup \bigcup\{Ev(p[\vec{n}/\vec{x}, \vec{k}/\vec{\chi}, \vec{L}/\vec{\psi}])\}$
Where $\vec{n}$ names, $\vec{k}$ are keys, and $\vec{L}$ are closed messages
$Ev(\|_{i\in I} p_i) \quad = \quad \bigcup_{i\in I} i : Ev(p_i)$
where, E is a set, and $i : E$ denotes the set $\{i : e \mid e \in E\}$.

2.1.7 General Proof principles

Verifying security properties in SPL is not as tedious as in other calculi since, its inherent proof techniques are based on its own operational principles. In other words, SPL uses its event based semantics to derive some general proof principles, which capture the notion of dependency between events in a protocol run. These principles, are of the essence of SPL's proof techniques but they are not the only concepts used for aiding the properties' verification. The proofs are simplified by a result of the occurrence of the spy events in the protocol run. The result is based on the notion of surroundings of a message inside another. These ideas inherent to the calculus are the ones used to verify or contradict the fulfilment of any security property in a protocol run.

From the net semantics we can derive several principles useful in proving authentication and secrecy of security protocols. Write $M \sqsubseteq M'$ to mean message M is a subexpression of message M', i.e., $\sqsubseteq$ is the smallest binary relation on messages st:

$$\begin{array}{lll} M \sqsubseteq M & & \\ M \sqsubseteq N & \Rightarrow & M \sqsubseteq N, N' \text{ and } M \sqsubseteq N', N \\ M \sqsubseteq N & \Rightarrow & M \sqsubseteq \{N\}_k \end{array}$$

where M, N, N' are messages and k is a key expression. We also write $M \sqsubset t$ iff $\exists M'.M \sqsubset M' \wedge M' \in t$, for a set of messages t.

Below we present a set of general proof principles strongly based on the work done by Federico Crazzolara in [Cra03].

Definition 1 (Well-foundedness.). *Given a property P on configurations, and $P(p_0, s_0, r_0)$ represents that configuration $\langle p_0, r_0, s_0 \rangle$ holds property P, if a run $\langle p_0, s_0, t_0 \rangle \xrightarrow{e_1} ... \xrightarrow{e_r} \langle p_r, s_r, t_r \rangle \xrightarrow{e_{r+1}} ...$, contains configurations st $P(p_0, s_0, t_0)$ and $\neg P(p_j, s_j, t_j)$, then there is an event e_h, $0 < h \leq j$, st $P(p_i, s_i, t_i)$ for all $i < h$ and $\neg P(p_h, s_h, t_h)$.*

We say that a name $m \in N$ is *fresh* on an event e if $m \in e^n$ and we write $Fresh(m, e)$

Definition 2 (Freshness.). *Within a run*

$$\langle p_0, s_0, t_0 \rangle \xrightarrow{e_1} ... \xrightarrow{e_r} \langle p_r, s_r, t_r \rangle \xrightarrow{e_{r+1}} ...,$$

the following properties hold:

1. *If $n \in s_i$ then either $n \in s_0$ or there is a previous event e_j st $Fresh(n, e_j)$.*
2. *Given a name n there is at most one event e_i st $Fresh(n, e_i)$.*
3. *If $Fresh(n, e_i)$ then for all $j < i$ the name n does not appear in $\langle p_j, s_j, t_j \rangle$.*

Definition 3 (Control Precedence.). *Within a run*

$$\langle p_0, s_0, t_0 \rangle \xrightarrow{e_1} ... \xrightarrow{e_r} \langle p_r, s_r, t_r \rangle \xrightarrow{e_{r+1}} ...,$$

if $b \in {}^c e_i$ either $b \in Ic(p_0)$ or there is an earlier event e_j, $j < i$, st $b \in e_j^o$.

Definition 4 (Output-input Precedence.). Within a run

$$\langle p_0, s_0, t_0 \rangle \xrightarrow{e_1} ... \xrightarrow{e_r} \langle p_r, s_r, t_r \rangle \xrightarrow{e_{r+1}} ...,$$

if $M \in {}^o e_i$, then either $M \in t_0$ or there is an earlier event e_j, $j < i$, st $M \in e_j^o$

Definition 5 (Output Principle.). *Within a run*

$$\langle p_0, s_0, t_0 \rangle \xrightarrow{e_1} ... \xrightarrow{e_r} \langle p_r, s_r, t_r \rangle \xrightarrow{e_{r+1}} ...,$$

According to the message persistence in SPL, $\forall\, e_v$ in a run, $e_v^o - e_{v-1}^o$ are the new messages generated by event e_v

2.1.8 Message Surroundings

Given a pair of messages M and N the surroundings of N in M are the smallest submessages of M containing N under one level of encryption. So for example the surroundings of $Key(A)$ in

$$(A, \{B, Key(A)\}_k, \{Key(A)\}_{k'})$$

are $\{B, Key(A)\}_k$ and $\{Key(A)\}_{k'}$. If N is a submessage of M but does not appear under encryption in M then we take the surroundings of N in M to be N itself.
For example the surroundings of $Key(A)$ in

$$(A, \{B, Key(A)\}_k, Key(A))$$

are $\{B, Key(A)\}_k$ and $Key(A)$.

Let M and N be two messages. Define $\sigma(N, M)$ the surroundings of N in M inductively as follows:

$$\begin{array}{lcl}
\sigma(N, v) & = & \begin{cases} \{v\} & \text{if } N = v \\ \emptyset & otherwise \end{cases} \\
\sigma(N, k) & = & \begin{cases} \{k\} & \text{if } N = k \\ \emptyset & otherwise \end{cases} \\
\sigma(N, (M, M')) & = & \begin{cases} \{(M, M')\} & \text{if } N = M, M' \\ \sigma(N, M) \cup \sigma(N, M') & otherwise \end{cases} \\
\sigma(N, \{M\}_k) & = & \begin{cases} \{\{M\}_k\} & \text{if } N \in \sigma(N, M) \text{ or } N = \{M\}_k \\ \sigma(N, M) & otherwise \end{cases} \\
\sigma(N, \psi) & = & \begin{cases} \{\psi\} & \text{if } N = \psi \\ \emptyset & otherwise \end{cases}
\end{array}$$

2.1.9 Proving Security Properties with SPL

There are some general steps which must be followed in order to verify security properties under this framework.

Any security property wanted to be verified must be modelled in a formal way. This can be done in a very intuitive way, by means of the notions of message surroundings, which capture the most important concepts needed for representing security predicates. Afterwards, in order to fulfil the already formalised property, every event in the protocol must be verified. An adequate method for proving these properties over such different events is the contradiction mechanism, by which one states a simple supposition, such as the existence of an event in which the property is not achieved, and by means of the event dependency presented in the SPL language and the proof principles mentioned before, one tries to find that the event which must exist in order to broke the property, never happens along the protocol run.

3 Dynamic Reconfiguration Systems

The problem of dynamic reconfiguration of systems is inherent to a wide variety of problems, such power consumption networks [DGOR04], agent networks [PR99], and P2P systems [WHY$^+$04b]. The problem addresses the inconveniences present where a distributed and highly dynamic system need to modify the states of each agent without loss of information. In this section, we explain in deeper detail this problem based in a specific problem of P2P systems: The reconfiguration of applications in P2P systems.

3.1 FTN protocol

The Friends Troubleshooting Network (FTN) is a protocol that explores the advantages of P2P approach in automatic reconfiguration of applications [WHY$^+$04b]. Placing in context, the protocol operates in an open environment where the correct behaviour of each agent depends on a configuration table, where stored entries are comformed by a key attribute and a privacy sensitive record value.

Basically the protocol sends the request of an misconfigured application and the suspicious entries that possibly origin the problem to a group of trusted agents (friends), they contribute to solve the problem revising its own records in search for suspects according the request and updating the vector of suspects modifying the probability for each suspect, as well including their own suspects. The protocol continues in the way that each friend could request for aid to his own friends, spreading the process until a fixed number of agents has collaborated in the request. Finally, each friend involved in the protocol returns backward the vector of suspects until the requester is reached, and he only has to repair the suspect entry with more probability.

There are several security aspects that we have to consider: First, relating to integrity, we must ensure that nobody can alter the contents of a given message. Second we must guarantee that nobody can trace the origin of the request. Third, that nobody can guess which entries are included or modified for some agent. and finally: that only the agents that are trusted must include information in the request.

3.1.1 Agent Definition

An agent in the P2P system can be either a *sick* machine, a *helper* or a *forwarder*. Each one of these roles is explained next.

Sick Machine The first step to make a request for the sick machine is to convert the privacy sensitive information (e.g., login/password information, credit card numbers, and so on) into widely known constants that preserve the semantics of the message.Then the requester must send to one of the trusted friends the request including the vector of suspicious entries mapped before, the name of the misconfigured application, a new name to identify the request, and the number of hops (network jumps) needed to end the search. Finally the sick machine is await for confirmation of the friend, if a confirmation message is received, the protocol simply waits for the eventually response of his friend and operates consequently, subtracting from the vector the value with most probability. Otherwise he chooses another friend and repeat the process.

Friend Machine The first thing that a friend agent do after receiving the request information, is to choose whether to help or not to the requester. This is done by sending the respectively acknowledge to the requester. The next step is to decide what role the friend is going to take in the protocol: to help modifying and including information into the vector of suspicious entries, or to forward the request to another friend. If he want to help in the request, the friend operates over the vector of suspicious entries adding its own suspects and incrementing values into previous entries, based on his local reasoning. The main aspect in this process is to help to sick machine without revealing its own applications. Finally, the friend verifies if it is the last hop in the protocol, sending the message forward if there are remaining hops waiting, or backward if is the last agent in the protocol.

Forwarder The forwarder simply selects one of his own friends and pass away the request, expecting their response for a limited time, and if it arrives send it backwards, otherwise he must cancel forward requests and send the trace to the previous agent in the protocol.

An example of the protocol are illustrated in Figure 2, where a sick machine S publish his request to his friends H_1 and F_1 which are intended to participate in the request helping and forwarding the data. Each agent that helps in the request includes information to the vector of suspect entries (as seen in the output messages of H_1, H_2 and H_4). If an agent has already collaborate in a request, it stops the input request (denoted as a dotted line between F_1 and F_4). Also, the protocol ends when a fixed number of collaborative agents are involved in the protocol (in this case, the value are limited by 3) sending the response backwards, so other traces that cannot reach this level will be stuck and the friend agents can never reply to the sender its values.

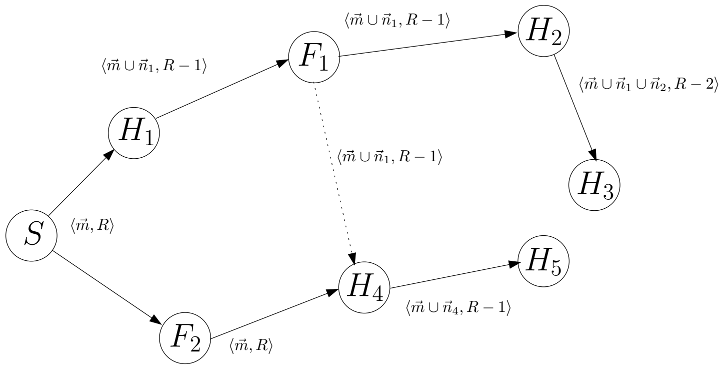

$\langle \vec{m} \cup \vec{n}_1, R-1 \rangle$
$\langle \vec{m} \cup \vec{n}_1, R-1 \rangle$
$\langle \vec{m} \cup \vec{n}_1 \cup \vec{n}_2, R-2 \rangle$
$\langle \vec{m} \cup \vec{n}_1, R-1 \rangle$
$\langle \vec{m}, R \rangle$
$\langle \vec{m} \cup \vec{n}_4, R-1 \rangle$
$\langle \vec{m}, R \rangle$
S
H_1
F_1
H_2
H_3
H_4
H_5
F_2

Figure 2: Friends Troubleshooting Network Model

3.1.2 Known Attacks

This version of the protocol evidence two types of attacks which are covered broadly by Wang et al. at [WHY+04a], the first of them, called *Gossip Attack*, shows that a collusion between two non-immediate agents in the protocol could infer what are the new messages posted for the agent in the middle, as shown in figure 3.

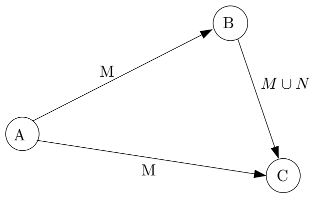


Figure 3: Gossip Attack: C could infer the contents added by B

Another attack that could break the secrecy of the messages in the protocol is *the polling attack*. This attack could make use of the parameter denoting the number of remaining hops needed to discover the secrets added by the last agent for the previous agent involved in the protocol.

3.2 Characteristics of Fixed FTN

With this considerations in mind, a new version of FTN was released. the main characteristic of the protocol fix include the concept of shared spaces: each helper that want to contribute into the protocol, must create a cluster with his own friends, sharing the messages of the request and modifying or publishing his own request into the cluster.

The procedure for the cluster is explained as follows. First, the cluster entrance B receives the message M, then it communicate M to his friends in order to establish the cluster. When the cluster is properly established, B publish M in the shared space and the agents in order that his friends could have access to the request. In this way the cluster members could publish the results of their own consults using M. Every agent in the system is a trusted friend so we can say that the information of the cluster is not used for his own purposes, and the number of messages included by each agent in the cluster depends of its own local computation. Finally, one of the cluster members forward the messages contained in the cluster in order to continue with the protocol. The image 4 illustrates the process above.

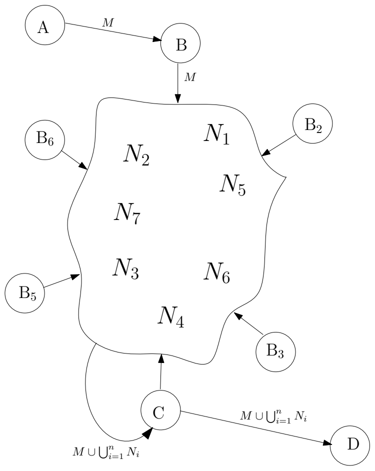


Figure 4: Cluster Modelling

Another of the corrections included in the revision of FTN was to change the number of hops, so no agent could know where is the last hop in the network. This could be made adding probability to the protocol changing R to $1 - \frac{1}{N}$ where N are the minimum number of samples needed, and the stop condition is modified so it stops when the probability $P(1 - \frac{1}{N}) \approx 0$

4 Security properties to be Assured

In this model, we must describe the security properties in order to prove the correctness and functionality of the protocol:

- **Integrity:** This property states that contents in a message must persist all over the life cycle of the message delivery. This means that any kind of information can be added to the message , but without altering its old contents. More formally, for every message response M' in transit from peer A to B the integrity of this message is ensured if $M \sqsubseteq M'$ such that M' is the message generated just before M. In this way we ensure a monotonic message, which is always part of the next generated message. Due to the importance of the answers from the agents involved in the request, we must ensure that:
  - Every data included by a friend peer into the answer, must remain until reaching the protocol requester agent.
- **Secrecy:** Also known as *Anonymity beyond suspicion*[MKL$^+$02]. Ensures that the real information published by an agent can never be known by other peers in the network, different

from its target. Formalising, for every message going from A to B, the information published is never showed as a cleartext, or as cyphertext which can be decrypted by other peers rather than the both mentioned before, during the delivery life cycle. In this way, we must show that:

– The plain text m created by an initiator agent A can never be derived from other messages in the protocol.

– The plain text x created by a friend agent B can never be derived from other messages in the protocol.

5 A close FTN approach with SPL

As we have explained, this protocol includes several concurrency considerations that involve security, such as the exclusive choice of roles, cluster handling, and mutable spaces in the protocol. These features are not defined in SPL, basically relying limitations concerned of the inherent model of persistent store. However, this class of constructions are widely provided and used in other process calculi such as π [Mil99] or Spi[AG97]. In this section we provide a set of encodings to achieve these task, formalising FTN network architecture as a well grounded example where this concepts remains crucial

5.1 Encodings

5.1.1 Exclusive Non-deterministic choice

The choice between two excluding processes is not a new idea, this operator was introduced by Milner [Mil99], Abadi & Fournet [AF01] and Palamidessi & Valencia [PV01], and intends to represent the execution of a process with tasks with the same possibility of being executed, differing from the parallel composition in the way that if one of them is selected, the other process remains stuck stopping their evolution over time. However, the concept of parallel composition, new nonces, and message exchange can serve as well for achieving this task, for example, given a process R with two exclusive choices P and Q:

- A public key f is generated and distributed to P and Q in order to guarantee the freshness of the choice.
- Both processes generate a fresh public key that is sent to a common process which selects one key according to the time of arrival, responding with a fresh name encrypted with the public key received.
- The process receiving the response will be the one which will execute, while the other will remain stuck forever.

Clearly, if a third process R is involved in a sequential composition, it has to wait until one of the process is completely executed. With the considerations presented before we present the formal model of this construction in SPL:

$$\begin{aligned}(P\,+\,Q).R \quad &\equiv out\,new(f,g)\{Pub(f)\}_{Pub(g)}(out\,new(s)\,\{Pub(s)\}_{Pub(f)}\,.\,in\,\{x\}_{Pub(s)}\,.\,P.R\,\|\\ &out\,new(t)\,\{Pub(t)\}_{Pub(f)}\,.\,in\,\{x\}_{Pub(t)}\,.\,Q.R\,\|\\ &in\,\{Pub(Z)\}_{Pub(f)}\,.\,out\,new(a)\,\{a\}_{Pub(Z)})\end{aligned} \tag{1}$$

5.1.2 Indexed Exclusive non-deterministic choice

A non-deterministic choice behaviour over a set of process P can be generalised from the previous encoding in the following way:

$$\begin{aligned}(\|_{+\,i\in\{1..n\}}P_i).R \equiv\, & out\,new(f,g)\{Pub(f)\}_{Pub(g)}.(\|_{i\in\{1..n\}}out\,new(s)\,\{Pub(s)\}_{Pub(f)}\,.\,in\,\{x\}_{Pub(s)}\,.\,P_i.R)\,\|\\ & in\,\{Pub(Z)\}_{Pub(f)}\,.\,out\,new(a)\,\{a\}_{Pub(Z)}\end{aligned} \tag{2}$$

It relies in the same concepts stated in 5.1.1

If a process R has to be executed strictly after an indexed non deterministic choice $\|_{+\,i\in\{1..n\}}P_i$ we adopt the same idea as in equation 1.

5.1.3 Indexed sequential composition

SPL presents an indexed parallel composition process by which represent several indexed processes working in parallel. Despite being a very important concept for concurrency, sometimes the need of ensuring that all processes will execute one after another and not at the same time arises. For example, taking a subtle modified version of the Readers and Writers mutual exclusion problem [CHP71]. In our own instance of the problem, every writer executes his task before the execution of the reader. In this particular case, the only problem arises when two or more writers want to modify the shared resource at the same time. Therefore, since every writer must execute its job having exclusive access to the critical section, we must ensure some kind of order in the set, in a way that while some agent is writing, the others just wait for their turn. A simple sequential composition between writing processes is not an adequate solution, due to the amount of processes that should be written in order to complete the whole composition, so we must make use of the new concept of indexed sequential composition.

Therefore, we will make some minimal changes to the parallel composition in such a way that we can turn it into a sequential composition.

$$\begin{aligned}\|_{seq\,i\in\{1..n\}}P_i \quad\equiv\quad & out\,new(a)\,\{a\}_{Key(P_0,P_1)}\,.\,(\|_{i\in\{1..n-1\}}\,in\,\{x\}_{Key(P_{i-1},P_i)}\,.\,P_i\,.\,out\,\{x\}_{Key(P_i,P_{i+1})})\,.\\ & in\,\{x\}_{Key(P_{n-1},P_n)}\,.\,P_n\end{aligned} \tag{3}$$

Explanation In this encoding, the key factor are the shared keys between the components inside the parallel composition. These keys will work as channels by which the indexed elements will communicate in a way that each one will trigger the execution of the other.

- We have an output process outside the parallel composition which will start the execution of the indexed processes. This can be easily seen because, for the first indexed process to get started, it first has to receive an acknowledge through the channel it shares with the initial process outside the parallel composition.
- In the same way, after the first process inside the parallel composition executes, it will send an acknowledge via the channel it shares with the following process and this one will have to wait until receiving it.
- The last component works outside the parallel composition. It awaits until receiving the acknowledge from the last process, to get started.

5.1.4 Sequential replication

In the same way as a Replication is an infinite parallel composition of processes, a sequential replication is an infinite indexed sequential composition of processes. This kind of process is needed when a processes must be executed infinitely, one after the other.

$$!_{seq}P \quad \equiv \quad out\,new(a)\,\{a\}_{Key(P_0,P_1)} \cdot \|_{i\in\{1..\infty\}}\, in\,\{x\}_{Key(P_{i-1},P_i)} \cdot P_i \cdot out\,\{x\}_{Key(P_i,P_{i+1})} \tag{4}$$

Explanation It relies in the same concepts stated in 5.1.3

5.2 Modelling a Cluster for FTN

Since SPL has a monotonic store, which means that messages output into the network persist forever, it turns to be really difficult to model a cluster by means of this language, requiring an space with mutable capacity . Then, modelling an abstraction of a mutable space on this calculus must be done via the encodings stated in 5.1. A mutable cluster in SPL can be seen as a store with several instances through its life time. Therefore, we model a store in which each time the messages of the cluster are modified, another instance is created, with a different and new lock which will identify the store, denying the access from intruders. The keys and locks to the space of messages will be managed by a the cluster initiator, updating the keys and redirecting the spaces each time the cluster is modified, assigning a turn of each of the principals involved. The cluster is a composition of two main processes, Initiator and Participants.

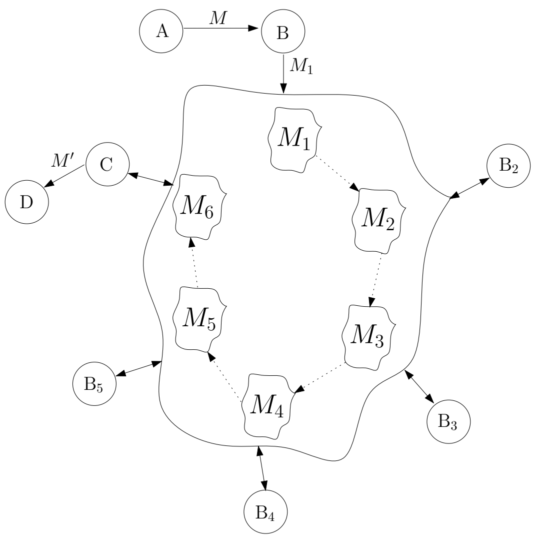


Figure 5: Cluster over a persistent network : The store evolves by means of linked stores

5.2.1 Initiator

This process initiates the cluster by generating its first instance. Following, it triggers the execution of the next component participating in the cluster.

$$\begin{aligned} Initiator(A,B,M) \quad \equiv \quad & out\,new(k)\,\{k\}_{Pub(A)} \cdot fun(A, Pub(k), M) \,. \\ & in\,\{M'\}_{Pub(k)} \cdot out\,new(a)\,\{M'\}_{Pub(a)} \cdot out\,\{Priv(a)\}_{Key(A,B)} \end{aligned} \tag{5}$$

Where A represents the cluster initiator which works as the keeper or manager of the cluster, M the message by which the first information of the cluster is generated and B the next component participating in the cluster. $fun(A, Pub(k), M)$ will represent the function by which the agent A generates a message M' by computing the already received message M with its own local information in a single tuple. The tuple M' generated by this function must include the contents of M. Finally the message M' is output to the space of messages encrypted with $Pub(k)$.

- Here A generates the first key and lock for the cluster, introducing the initial information inside the cluster, generated by means of the function stated before.
- Then A sends the key to the next participant B.

5.2.2 Participants

This process models the behaviour of the rest of agents participating in the cluster. It represents the way in which each peer interacts with the cluster by collaborating or not collaborating with the cause.

$$\begin{aligned} Participants(A, \vec{P}, n) \quad \equiv \quad & \|_{seq\, v \in \{1..n\}} \, (in\{Priv(Z)\}_{Key(A,P_v)} \,.\, in\,\{M\}_{Pub(Z)} \,. \\ & (Contributor(P_v, A, M, Priv(Z)) + Non - Contributor(P_v, A, Priv(Z))) \,. \\ & Distributor(A, P_v)) \end{aligned}$$

where

$$\begin{aligned} Contributor(A, B, M, Priv(Z)) \quad & \equiv \quad out\, new(k)\, \{k\}_{Pub(A)} \,.\, fun(Pub(k), A, M) \,.\, in\{M'\}_{Pub(k)} \,. \\ & \qquad out\, new(b)\, \{M'\}_{Pub(b)} \,.\, out\,\{Priv(b), Priv(Z)\}_{Key(B,A)} \\ Non - Contributor(A, B, Priv(Z)) \quad & \equiv \quad out\,\{Priv(Z), Priv(Z)\}_{Key(B,A)} \\ Distributor(A, P_m) \quad & \equiv \quad in\,\{Priv(X), Priv(Y)\}_{Key(A,P_m)} \,. \\ & \qquad out\,\{Priv(X)\}_{Key(A,P_{m+1})} \end{aligned}$$

Figure 6: Cluster Formalisation

Where A is the cluster manager, $\vec{P}$ the rest of friends participating in the cluster and n the cardinality of $\vec{P}$.

- The first agent receives key $(Priv(Z))$ and opens the store for the information inside it.
- If this peer does not want to modify any content inside the cluster, it just executes $Non - Contributor$ and sends back the same key to the server which will pass it to the next process. But, if the agent wants to modify the contents of the cluster it executes the $Contributor$ process, by which it generates a new key $(Priv(a))$ and lock $(Pub(a))$ and calls the function $fun(P, Pub(k), M)$ by which M' is generated. The agent receives M' encrypted with $Pub(k)$, decrypts it and locks it with its previously generated lock, $(Pub(a))$. Then, it sends the key $(Priv(a))$ to the server which will continue, and forward that new key to the next participant in the cluster by means of the $Distributor$ process.

Putting all together, the cluster can be formalised as:

$$Cluster(S, \vec{P}, n, init) \quad \equiv \quad Initiator(S, P_1, init) \,.\, Participants(S, \vec{P}, n) \tag{6}$$

5.3 Assumptions

In order to model the FTN protocol among these encodings, we have to include some assumptions for the reader's understanding. We focus on the modelling of anonymous communications in a well established network, so we consider a model where authentication between peers was previously done using an authentication protocol. Let $Peers(G)$ represent the whole P2P network and $f(S)$ the set of friends of any agent S.

Definition 6 (FTN Messages)**.** Let $I \equiv (Rid, init)$ an initial message, where Rid is a message identifier and $init$ a tuple which will include all the information required for the initiator to request some help. The initial message I evolves through the protocol in the following way $I \rightarrow I' ... \rightarrow L$, where $I' \equiv (Rid, (init, M))$ and M is the new information added by each friend in the protocol, which decides to help the requester. An finally $L \equiv (Rid, (init, M), end)$ represents the last message sent back to the initiator via the same path where it arrived. In this last message the name end, known by every peer in the network, is included to identify this specific message as the one which has to be sent backwards, until reaching the requester process.

5.4 Requester behaviour

The requester or initiator A generates a message with the following structure:

$$\{Rid, init\}_{Key(A,X)} \text{ Where } X \in f(A)$$

In this way the requester sends the message of all of its friends, which will decide if the will help it or will just forward its request.

- Request Output: $out\, new(Rid, init)\, \{Rid, init\}_{Key(A,X)}$ where $X \in f(A)$. The output request will be sent to every friend of the requester, encrypted with shared key between friends, in such a way that the only one which can understand the message are the group of friends of the initiator.
- Reception of the answer: $in\, \{Rid, (init, M), end\}_{Key(Y,A)}$. The requester receives as an answer the first message received by one of its friends, including the name end, which will mean that the data recollection have ended, and the message now includes the information required for the solution of its problem.

This behaviour is condensed in figure 7: (Recalling the message structure presented in definition 6)

$$Init(A) \quad = \quad (\|_{i \in f(A)}\, out\, new(Rid, init)\, \{I\}_{Key(A,i)}) \,.\, in\{L\}_{Key(A,i)}$$

Figure 7: Model of a Requester

5.5 Helper Agent Behaviour

The first action that a friend has to resolve is to help or to forward, If the agent decide to help, it generates a cluster with a group of trusted friends in such a way that inside this store, a great amount of information can be recollected. Then, the helper selects one of the principals involved in the cluster and pass the control over the information received in the cluster, one of the friends sharing the cluster, takes out the last information remaining. This agent has two similar options to take, either it may forward this information to another friend which will decide to help or not, or it can just send back the recollected data to the the originator of the cluster, in a way that it eventually the protocol initiator can be reached.

- Decision: The agent which takes out the information from the cluster, has to decide between continue helping (by forwarding the data took out from the cluster), or just sending back the information to the cluster initiator, which will redirect it until it reaches the protocol initiator. This is done in a non-deterministic way by means of the choice encoding: $HelperFwd(A) + HelperBckwd(A)$.

- Reception of the request: $in\{Rid, M\}_{Key(Y,A)}$ Here the helper receives the message capturing the information needed to proceed, with its help in the variable M.

- Cluster Help: $Cluster(S, f(A), n, (Rid, M))$. Here the helper generates a cluster by which it will recollect information to help the initiator of the process. The helper always acts as manager S which will be in charge of the cluster.

- Continue helping: In the process $HelperFwd(A)$ a chosen helper takes out the information from the cluster and decides to forward the message to another friend, waiting for a response which will send back to the cluster initiator.

- Sending Back: In process $HelperBckwd(A)$ the chosen helper takes out the data from the cluster and sends back the information to the cluster initiator which redirects it back.

With this considerations, the helper is modelled in figure 8

$$\begin{array}{lcl}
HelperFwd(A) & = & (\|_{i\in f(f(A)_n)}\ out\,\{I'\}_{Key(f(A)_n,i)})\,.\,in\,\{L\}_{Key(f(A)_n,i)} \\
 & & out\,\{L\}_{Key(f(A)_n,A)}\,.\,in\,\{L\}_{Key(f(A)_n,A)}\,.\,out\,\{L\}_{Key(A,Y)} \\
 & & \\
HelperBckwd(A) & = & out\,\{L\}_{Key(f(A)_n,A)}\,.\,in\,\{L\}_{Key(f(A)_n,A)}\,.\,out\,\{L\}_{Key(A,Y)} \\
 & & \\
Helper(A) & = & Collaborate(A).\,(HelperFwd(A)\,+\,HelperBckwd(A)) \\
\text{Where} & & \\
Collaborate(A) & = & \|_{Y\in f(X)}\ in\,\{I'\}_{Key(Y,A)}\,.\,Cluster(A, f(A), n, I')\,.\,in\,\{I'\}_{Pub(x)}
\end{array}$$

Figure 8: Model of a Helper

5.6 Forwarder Role

- Forwards the request and waits for the response in order to return it to the sender. The model of this agent is shown in Figure 9 (We recall the message structure presented in definition 6)

$$Fwd(A) \quad = \quad !in\,\{I'\}_{Key(Y,A)}\,.\,(\|_{i\in f(A)}\,out\,\{I'\}_{Key(A,i)})\,.\,in\,\{L\}\,.\,out\,\{L\}_{Key(A,Y)}$$

Figure 9: Model of a Forwarder

$$\begin{aligned} Node(A) &= Init(A)\,\|\,Fwd(A)\,\|\,Helper(A) \\ FTN &= \|_{A\in Peers(G)}\,Node(A) \end{aligned}$$

Figure 10: Instance of FTN Protocol

5.7 The FTN Protocol

Putting all together, the instance of the protocol is modelled below:

Here we have the FTN protocol, where the initiator is the sick machine which wants to be helped. It sends a collaboration message to all its friends, which will either help it, or forward the request in their own behalf, to one of their friends. If the friend of the initiator or just a subsequent friend wants to help, it will call all its own friends and will organise a cluster. There, all participants will make a brainstorm and will recollect information which can be sent back to the initiator through the same path, or could be moved forward in a search for more information.

6 Dynamic Reconfiguration Protocol: an FTN simplified protocol

In this model, we pretend to conserve the functionality of the system and the main security properties with a model strictly close to SPL, with a much more simpler protocol. The Dynamic Reconfiguration protocol (DR), modifies the way each agent interacts, with ideas inspired in multiple encryption stages, as in the Onion routing protocol [GRS99]. In this way, we abstract certain aspects of the protocol, like the use of an anonymising function, in order to fulfill the requirements imposed. We will represent a P2P network using the Dolev-Yao model.

The intuitive description of the protocol is presented below:

$$\begin{array}{ll} A \longrightarrow X: & \{R, Pub(k), \{M\}_{Pub(k)}\}_{key(A,X)} \text{ where } X \in f(A) \\ X \longrightarrow Y: & \{R, Pub(k), \{\{M\}_{Pub(k)}, P\}_{Pub(k)}\}_{key(X,Y)} \text{ where } Y \in f(X) \\ \ldots & \\ Y \longrightarrow B: & \{R, Pub(k), \{M', P\}_{Pub(k)}\}_{key(Y,B)} \text{ where } B \in f(Y) \\ B \longrightarrow X: & \{N, R, Pub(k), M'\}_{key(B,X} \text{ where } X \in f(B) \\ X \longrightarrow Y: & \{N, R, Pub(k), M'\}_{key(X,Y)} \text{ where } Y \in f(X) \\ \ldots & \\ Y \longrightarrow A: & \{N, R, Pub(k), M'\}_{key(Y,A)} \text{ where } A \in f(Y) \end{array}$$

Figure 11: Dolev-Yao Model of the DR protocol

In this scheme, the initiator agent A creates a request, with a new identifier Rid, a new public key $Pub(k)$ and a new secret $\{M\}_{Pub(k)}$. It sends the request, encrypted with a shared key $Key(A, P_1)$ to a friend agent P_1. In this way, P_1 receives the information sent by A and includes into the request his own information, ciphering it with the public key sent in the request. This process is made for each agent present in the protocol, constructing a ciphered-layer message, only possible to discover for the owner of the key (A in this case). This process continue until the last helper agent in the protocol includes its own information in the request, sending back the response message using the same path where he had received the request, ciphering the message with key $Pub(k)$. Finally, A receives the message and recurrently decrypts the message until it reaches his own genererated identifier nonce, verifying the integrity of the information if the secret is inside the response..

6.1 DR Formalisation

Definition 7 (Layered Messages)**.** Every message ψ in the DR protocol has the following shape: $\psi \in \{\{m\}_{Pub(k)}, \{\{m\}_{Pub(k)}, p\}_{Pub(k)}, \{\{\{m\}_{Pub(k)}, p\}_{Pub(k)}, p\}_{Pub(k)}, ...\}$ Where m is the variable in which the nonce identifier generated by the request should go, $Pub(k)$ is the public key generated by the initiator of the protocol and p is the variable in which the information included by each helper should remain.

Definition 8 (Submessages under any level of encryptions)**.** Let $\psi\langle x\rangle$ be a message where $x \gg \psi$. Where $x \gg \psi$ is a relation defined in the following way: $x \gg \psi$ if $x \sqsubseteq \psi \vee \exists\psi_0$ st. $\psi_0 \sqsubseteq \psi \wedge \psi_0 \neq \psi \wedge x \gg \psi_0$

Definition 9 (Encryption Seed)**.** Let $\psi\langle x\rangle[x/m]$ a message where x appears under any level of encryptions but just substituting the m variable inherent to the message shape.

Definition 10 (FTN Sets)**.** Let $Info$ represents the data owned by all peers in the network, $Info(X)$ the information belonging specifically to peer X., $f(X)$ represents the set of friends of peer X, and $Peers(G)$ the set of all peers in the network. In our model we assume that $Key(X, Y) = Key(Y, X)$

$$\begin{array}{lcl}
Alice(X) & \equiv & (\|_{i\in f(X)}\ out\,new\,(Rid, k, m) \left\{Rid, Pub(k), \{m\}_{Pub(k)}\right\}_{Key(X,i)}) \\
 & & .\,in \left\{n, Rid, Pub(k), \psi\langle\{m\}_{Pub(k)}\rangle\right\}_{Key(i,X)} \\
Bob(X) & \equiv & !in\ \{res\}_{Key(Y,X)}\ (Fwd(X, Y, res) \parallel Triumph(X, Y, res)) \\
Node(X) & \equiv & Alice(X) \parallel Bob(X) \\
DR* & \equiv & \|_{X\in Peers(G)}\ Node(x)
\end{array}$$

Where

$$\begin{array}{lcl}
Fwd(X, Y, res) & \equiv & (\|_{j\in f(X)}\ out \left\{Rid, Pub(k), \left\{\{m\}_{Pub(k)}, p\right\}_{Pub(k)}\right\}_{Key(X,j)}) \\
 & & .\,in\,\{n, res\}_{Key(j,X)}\ .\,out\ \{n, res\}_{Key(X,Y)} \\
Triumph(X, Y, res) & \equiv & out\,new\,(n) \left\{n, Rid, pub(k), \{\psi, p\}_{Pub(k)}\right\}_{Key(X,Y)}
\end{array}$$

And

$$res \equiv (Rid, Pub(k), \psi)$$

Figure 12: SPL model of DR protocol

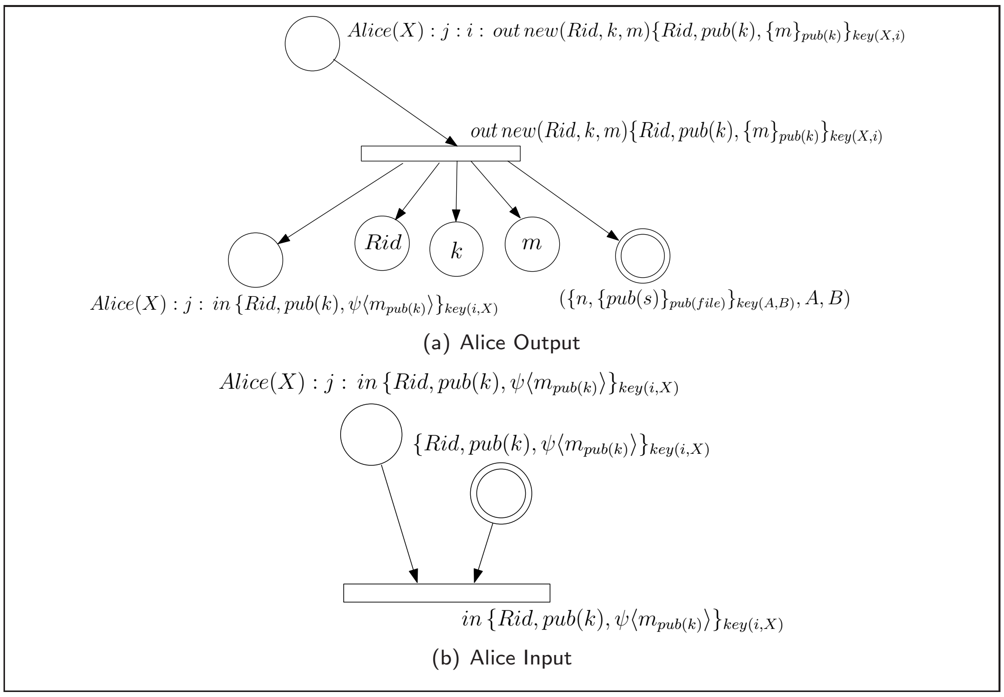
The protocol consists of an interaction between two kind of processes, $Alice(X)$ and $Bob(X)$. $Alice(X)$ is declared as an initiator agent that first creates the request identifier Rid, and a new pair of names k, m, then sends the request message to his friends including the fresh name m encrypted with $Pub(k)$ among with Rid. Finally, the agent expects for a reception message with the responses p encrypted in a multilayer system, with all the layers ciphered using the public key of k, including the encrypted fresh name m sent previously and a new name n which identifies the message as an answer.

$Bob(X)$ denotes a friend agent that receives the request information and operates forwarding the response message with his own suspects to one of its friends, ciphering the tuple that contains the contents received previously and the new message in a new encryption layer with the public key of k. It also can send the multi-layered encryption response immediately to the initiator, among with a new name n which denotes that the message shall go back through the same path it came in. The concrete model of the DR protocol can be seen in figure 12

6.2 Events

6.2.1 Alice Events

Alice events represent the actions available for a general requester in the friends network. Alice is composed by two subprocesses : An output process (fig. 13(a)), where Alice sends a message $\{Rid, pub(k), \{m\}_{pub(k)}\}_{key(X,i)}$ requesting for help to any of her friends in $f(x)$, generating new names Rid, k, m. The second action available for Alice is the reception of an answer contained in the message $\{Rid, pub(k), \psi\langle\{m\}_{pub(k)}\rangle\}_{key(i,X)}$ via an action $in\,\{Rid, pub(k), \psi\langle\{m\}_{pub(k)}\rangle\}_{key(i,X)}$ (fig. 13(b)).


$Alice(X) : j : i : out\,new(Rid, k, m)\{Rid, pub(k), \{m\}_{pub(k)}\}_{key(X,i)}$
$out\,new(Rid, k, m)\{Rid, pub(k), \{m\}_{pub(k)}\}_{key(X,i)}$
Rid
k
m
$Alice(X) : j : in\,\{Rid, pub(k), \psi\langle m_{pub(k)}\rangle\}_{key(i,X)}$
$(\{n, \{pub(s)\}_{pub(file)}\}_{key(A,B)}, A, B)$
$Alice(X) : j : in\,\{Rid, pub(k), \psi\langle m_{pub(k)}\rangle\}_{key(i,X)}$
$\{Rid, pub(k), \psi\langle m_{pub(k)}\rangle\}_{key(i,X)}$
$in\,\{Rid, pub(k), \psi\langle m_{pub(k)}\rangle\}_{key(i,X)}$
(a) Alice Output

(b) Alice Input

Figure 13: *Alice* Events

6.2.2 Bob Events

An execution of the agent Bob can be branched in a number of sub-processes: the initial event done is the reception of a request message $\{Rid, pub(k), \psi\}_{key(Y,X)}$ from any of the friends in $f(A)$ via an input action $in\,\{Rid, pub(k), \psi\}_{key(Y,X)}$. At this point, Bob can evolve in one of the sub-process of forwarding or response transmission.

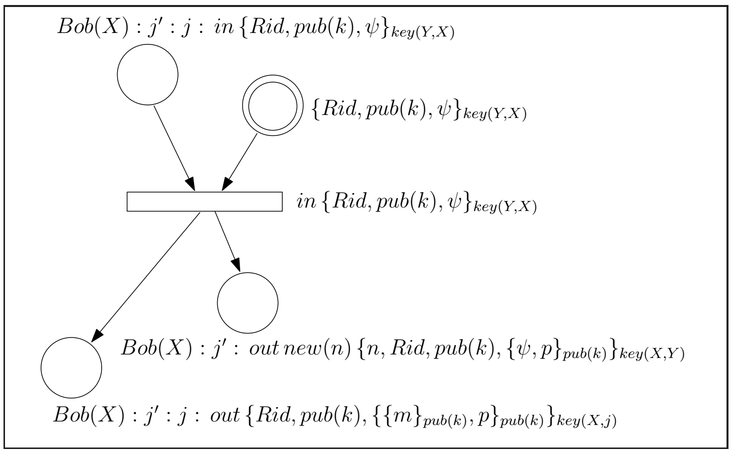

$Bob(X) : j' : j : in\,\{Rid, pub(k), \psi\}_{key(Y,X)}$
$\{Rid, pub(k), \psi\}_{key(Y,X)}$
$in\,\{Rid, pub(k), \psi\}_{key(Y,X)}$
$Bob(X) : j' : out\,new(n)\,\{n, Rid, pub(k), \{\psi, p\}_{pub(k)}\}_{key(X,Y)}$
$Bob(X) : j' : j : out\,\{Rid, pub(k), \{\{m\}_{pub(k)}, p\}_{pub(k)}\}_{key(X,j)}$

Figure 14: *Bob* Initial Event

6.2.3 Forwarder Events

Forwarder events indicate those events in which Bob helps contributing with the request and sending the modified message to a friend for further assistance. It is basically composed by three sub-processes: The first process (fig. 15(a)) generates an output event with the message $\{Rid, pub(k), \{\{m\}_{pub(k)}, p\}_{pub(k)}\}_{key(X,j)}$ via an output action $out\,\{Rid, pub(k), \{\{m\}_{pub(k)}, p\}_{pub(k)}\}_{key(X,j)}$. The next action available for the forwarder generates an input event for the answer messages $\{n, Rid, pub(k), \psi\}_{key(j,X)}$, sent back towards the originator agent *Alice* (fig. 15(b)). Finally, the last action (fig. 15(c)) generates an output event with the message $\{n, Rid, pub(k), \psi\}_{key(X,Y)}$ by means of the output action $out\,\{n, Rid, pub(k), \psi\}_{key(X,Y)}$.

6.2.4 Triumph Event

Bob Triumph event indicates the event in which the help ends, generating a message $\{\psi, p\}_{pub(k)}\}_{key(X,Y)}$ with a new name n, with the action $out\,new(n)\,\{n, Rid, pub(k), \{\psi, p\}_{pub(k)}\}_{key(X,Y)}$, as can be seen in figure 16

6.3 Definition of the Spy

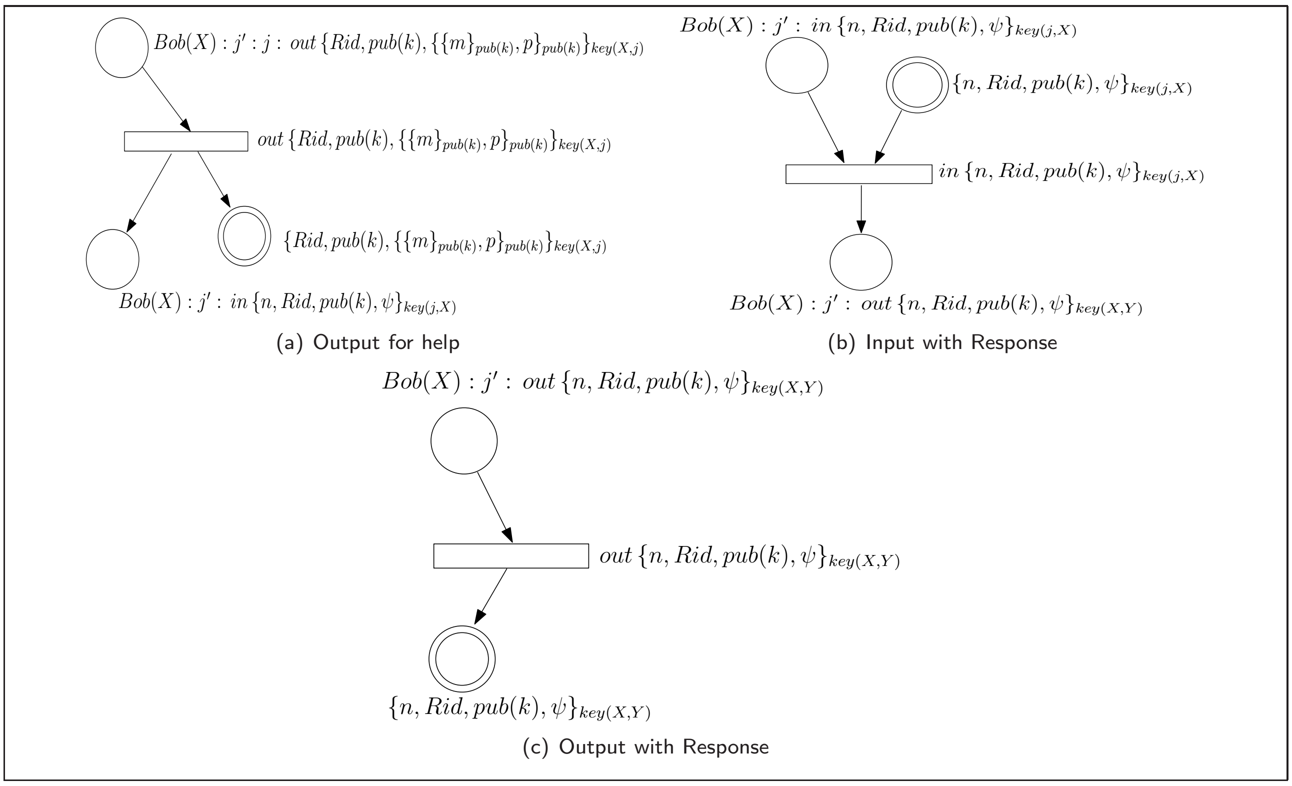
We use the definition of a powerful spy used in SPL [Cra03] to model the ways of intrusion and attack that an agent can do.

$$DR \equiv DR * \|!Spy$$

6.4 Secrecy Proofs in DR

To ensure the secrecy property for the response messages in the FTN protocol, we must follow a set of general steps.

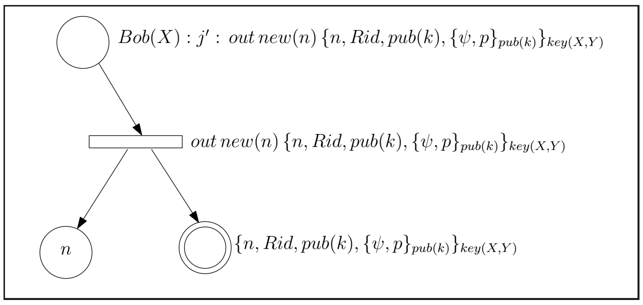
Initially, we must verify that the private keys used for encrypting the information added by each helper, are never leaked during message transmissions. This fact is relevant in order to assure that the data added by a friend who wants to help the sick machine, could be understood only by the initiator peer.


$Bob(X) : j' : j : out\,\{Rid, pub(k), \{\{m\}_{pub(k)}, p\}_{pub(k)}\}_{key(X,j)}$
$out\,\{Rid, pub(k), \{\{m\}_{pub(k)}, p\}_{pub(k)}\}_{key(X,j)}$
$\{Rid, pub(k), \{\{m\}_{pub(k)}, p\}_{pub(k)}\}_{key(X,j)}$
$Bob(X) : j' : in\,\{n, Rid, pub(k), \psi\}_{key(j,X)}$
$Bob(X) : j' : in\,\{n, Rid, pub(k), \psi\}_{key(j,X)}$
$\{n, Rid, pub(k), \psi\}_{key(j,X)}$
$in\,\{n, Rid, pub(k), \psi\}_{key(j,X)}$
$Bob(X) : j' : out\,\{n, Rid, pub(k), \psi\}_{key(X,Y)}$
$Bob(X) : j' : out\,\{n, Rid, pub(k), \psi\}_{key(X,Y)}$
$out\,\{n, Rid, pub(k), \psi\}_{key(X,Y)}$
$\{n, Rid, pub(k), \psi\}_{key(X,Y)}$
(a) Output for help

(b) Input with Response

(c) Output with Response

Figure 15: Forwarder Events


$Bob(X) : j' : out\,new(n)\,\{n, Rid, pub(k), \{\psi, p\}_{pub(k)}\}_{key(X,Y)}$
$out\,new(n)\,\{n, Rid, pub(k), \{\psi, p\}_{pub(k)}\}_{key(X,Y)}$
n
$\{n, Rid, pub(k), \{\psi, p\}_{pub(k)}\}_{key(X,Y)}$
Figure 16: *Triumph* Event

Then, assuming that those keys are never leaked, this secrecy property can be proved in a straightforward way, by presenting a stronger property which states that every response message added by a friend, is encrypted with a private key only known by the initiator of the protocol, and since we know that messages encrypted with these keys can never be decrypted by other rather than the peer requesting for help, the secrecy property for responses is fulfilled. In order to verify this property, each output event occurring in the protocol must be verified, to ensure that there is no message responses from friends intended for the initiator, which appear in non ciphered messages.

6.4.1 Secrecy Property for Private Key generated by the initiator

The first theorem for the DR protocol regards the private key generated by the initiator. If this private key is not corrupted from the start, and the nodes in the network behave as the protocol states, then this key will not be leaked during a protocol run. If we assume that $Priv(k) \not\sqsubseteq t_0$ where k is a name generated by the initiator, then at the initial state of the run there is no danger of corruption. This theorem will help us to prove some other security properties within the protocol.

Theorem 2. *Given a run of the DR and k_0 is a name generated by the requester, if $Priv(k_0) \not\sqsubseteq t_0$ then at each stage w in the run, $Priv(k_0) \not\sqsubseteq t_w$*

Proof. Suppose there is a run of DR in which $priv(k_0)$ appears on a message sent over the network. This means, since $Priv(k_0) \not\sqsubseteq t_0$, there is a stage $w > 0$ in the run st

$$Priv(k_0) \not\sqsubseteq t_{w-1} \text{ and } Priv(k_0) \sqsubseteq t_w$$

The event e_w is an event in the set

$$Ev(DR) \equiv Alice : Ev(p_{Alice}) \cup Bob : Ev(p_{Bob}) \cup Spy : Ev(p_{Spy})$$

and by the token game of nets with persistent conditions, is st

$$Priv(k_0) \sqsubseteq e_w^o$$

As can easily be checked, the shape of every $Alice$ or Bob.

$$e \in Alice : Ev(p_{Alice}) \cup Bob : Ev(p_{Bob})$$

is st

$$Priv(k_0) \not\sqsubseteq e^o$$

The event e_w can therefore only be a Spy event, if $e_w \in Spy : Ev(p_{Spy})$, however by control precedence and the token game , we would find an early stage u in the run, $u < w$ st $priv(k_0) \sqsubseteq t_u$ and therefore a contradiction is reached. □

6.4.2 Secrecy Property for the response help intended for the Requester

This theorem concerns the secrecy property for all responses p intended for the requester. It states that all the responses which flow through the network will never be visible for other peers different from the requester.

Theorem 3. *Given a run of DR st $X_0 \in Peers(G)$, $p_0 \in Info$, $Priv(k_0) \not\sqsubseteq t_0$ and the run contains a Bob event b_1 labeled with action*

$$act(b_1) = B : (X_0) : i_0 : j : out\{Rid_0, Pub(k_0), \{\psi, p_0\}_{Pub(k_0)}\}_{Key(X,j)}$$

Where i_0 is a session index, j is an index which belongs to the set $f(X)$, Rid_0 and k_0 are names and $Pub(k_0)$ is a public key associated to the name k_0, and $p_0 \in Info$. Then at every stage w $p_0 \not\sqsubseteq t_w$.

Proof. We show a stronger property such as this

$$Q(p, s, t) \Leftrightarrow \sigma(p_0, t) \subseteq \{\{n_0, Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,Y)}, \{\{Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,Y)}\}$$

If we can show that at every stage w of the run $Q(p_w, s_w, t_w)$ then clearly $p_0 \not\in t_w$ for all stages w in the run. Suppose the opposite statement, that at some stage in the run, property Q does not hold, by freshness clearly $Q(DR, s_0, t_0)$. Let v by well foundedness be the first stage in th run st $\neg Q(p_v, s_v, t_v)$. From the freshness principle it follows

$$a_1 \longrightarrow e_v$$

and from the token game of nets $\{Rid_0, Pub(k_0), \{\psi, p_0\}_{Pub(k_0)}\}_{Key(X,j)} \in \sigma(p_0, t_{v-1})$ (Because messages are persistent in the net). The event e_v is an event in

$$Ev(DR) \equiv Alice : Ev(P_{Alice}) \cup Bob : Ev(P_{Bob}) \cup Spy : Ev(P_{Spy})$$

and from the token game of nets with persistent conditions is st

$$\sigma(p_0, e_v^o - e_{v-1}^o) \not\subseteq \{\{n_0, Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,Y)}, \{\{Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,Y)}\} \quad (7)$$

Clearly e_v can only be an output event since $e_v^o - e_{v-1}^o = \emptyset$ for all input events e. Examining the output events of $Ev(DR)$ we conclude that $e_v \not\in Ev(DR)$ reaching a contradiction.

In the following lines we will explore each output event in the protocol in order to verify that the event e_v is different to all them.

Alice output events.

$$act(e_v) = Alice : (X) : j : i : out\,new\,(Rid, k, m)\{Rid, Pub(k), \{m\}_{Pub(k)}\}_{Key(X,i)}$$

Where $X \in Peers(G)$ and so $X \in s_0$, where Rid, m and k are names, $Pub(k)$ is a public key associated to the name k, j is a session index and i is an index which belongs to the set $f(X)$ where $i \in Peers(G)$ and so $i \in s_0$. Property 7 and the definition of message surroundings imply that $p_0 \gg \{Rid, Pub(k), \{m\}_{Pub(k)}\}_{Key(X,i)}$. From the freshness property $p_0 \neq Rid$, $p_0 \neq Pub(k)$ and $p_0 \neq m$. Therefore e_v can not be an A event with the above action.

Bob output events.

Case Fwd First output event

$$act(e_v) = Bob : (X) : j' : j : out\,\{Rid, Pub(k), \{\psi, p\}_{Pub(k)}\}_{Key(X,j)}$$

Where $X \in Peers(G)$ and so $X \in s_0$, where Rid, m and k are names, $Pub(k)$ is a public key associated to the name k, $p \in info$, j' is a session index and j is an index which belongs to the set $f(X)$ where $j \in Peers(G)$ and so $j \in s_0$. Property 7 and the definition of message surroundings imply that $p_0 \gg \{Rid, Pub(k), \{\psi, p\}_{Pub(k)}\}_{Key(X,j)}$. If $p_0 = p$ or $\psi\langle p_0\rangle$ then we reach a contradiction to property 7 because from the output principle it follows that $e_v^o - e_{v-1}^o = \{Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,j)}$. Then since property 7 must hold, $p_0 = Rid$ or $p_0 = Pub(k)$. By control precedence there exists an event e_u in the run st.

$$e_u \longrightarrow e_v$$

And

$$act(e_u) = Bob : (X) : j' : in\{Rid, Pub(k), \psi\}_{Key(Y,X)}$$

By the token game

$$\{Rid, Pub(k), \psi\}_{Key(Y,X)} \in t_{u-1}$$

and $\neg Q(p_{u-1}, s_{u-1}, t_{u-1})$ since $\{p_0, Pub(k), \psi\}_{Key(Y,X)} \in \sigma(p_0, t_{u-1})$ or $\{Rid, p_0, \psi\}_{Key(Y,X)} \in \sigma(p_0, t_{u-1})$ and then $\sigma(p_0, t_{u-1}) \not\sqsubseteq \{\{n_0, Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,Y)}, \{Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,Y)}\}$, a contradiction follows because $u < v$.

Case Fwd Second output event

$$act(e_v) = Bob : (X) : j' : out\,\{n, Rid, Pub(k), \psi\}_{Key(X,Y)}$$

Where $X \in Peers(G)$ and so $X \in s_0$, where n, Rid, m and k are names, $Pub(k)$ is a public key associated to the name k and j' is a session index. Property 7 and the definition of message surroundings imply that $p_0 \gg \{n, Rid, Pub(k), \psi\}_{Key(X,Y)}$. If $\psi\langle p_0\rangle$ then we reach a contradiction to property 7 because from the output principle it follows that $e^o_v - e^o_{v-1} = \{n_0, Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,Y)}$. Property 7 must hold then, $p_0 = n$ or $p_0 = Rid$ or $m_0 = Pub(k)$. By control precedence there exists an event e_u in the run st.

$$e_u \longrightarrow e_v$$

and

$$act(e_u) = Bob : (X) : j' : in\{n, Rid, Pub(k), \psi\}_{Key(j,X)}$$

By the token game

$$\{n, Rid, Pub(k), \psi\}_{Key(j,X)} \in t_{u-1}$$

and $\neg Q(p_{u-1}, s_{u-1}, t_{u-1})$ since $\{p_0, Rid, Pub(k), \psi\}_{Key(j,X)} \in \sigma(p_0, t_{u-1})$ or $\{n, p_0, Pub(k), \psi\}_{Key(j,X)} \in \sigma(p_0, t_{u-1})$ or $\{n, Rid, p_0, \psi\}_{Key(j,X)} \in \sigma(p_0, t_{u-1})$ and then $\sigma(p_0, t_{u-1}) \not\sqsubseteq \{\{n_0, Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,Y)}, \{Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,Y)}\}$, a contradiction follows because $u < v$.

Case $Triumph$ output event

$$act(e_v) = Bob : (X) : j' : out\,new(n)\,\{n, Rid, Pub(k), \{\psi, p\}_{Pub(k)}\}_{Key(X,Y)}$$

Where $X \in Peers(G)$ and so $X \in s_0$, where n, Rid, m and k are names, $Pub(k)$ is a public key associated to the name k, $p \in info$ and j' is a session index. Property 7 and the definition of message surroundings imply that $p_0 \gg \{n, Rid, Pub(k), \{\psi, p\}_{Pub(k)}\}_{Key(X,Y)}$. From the freshness principle, $p_0 \neq n$. If $p = p_0$ or $\psi\langle p_0\rangle$ we reach a contradiction to property 7 because from the output principle it follows that $e^o_v - e^o_{v-1} = \{n_0, Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,Y)}$. Then since property 7 must hold, $p_0 = n$ or $p_0 = Rid$ or $p_0 = Pub(k)$. By control precedence there exists an event e_u in the run st

$$e_u \longrightarrow e_v$$

and

$$act(e_u) = Bob : (X) : j' : in\{n, Rid, Pub(k), \psi\}_{Key(Y,X)}$$

By the token game

$$\{n, Rid, Pub(k), \psi\}_{Key(Y,X)} \in t_{u-1}$$

and $\neg Q(p_{u-1}, s_{u-1}, t_{u-1})$ since $\{p_0, Rid, Pub(k), \psi\}_{Key(Y,X)} \in \sigma(p_0, t_{u-1})$ or $\{n, p_0, Pub(k), \psi\}_{Key(Y,X)} \in \sigma(p_0, t_{u-1})$ or $\{n, Rid, p_0, \psi\}_{Key(Y,X)} \in \sigma(p_0, t_{u-1})$ and then $\sigma(p_0, t_{u-1}) \not\sqsubseteq \{\{n_0, Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,Y)}, \{Rid_0, Pub(k_0), \psi\langle p_0\rangle\}_{Key(X,Y)}\}$, a contradiction follows because $u < v$.

Spy output events An assumption of the theorem is that the private key of the requester is not leaked, meaning that $Priv(k) \not\sqsubseteq t_0$. At every stage w in the run $Priv(k) \not\sqsubseteq t_w$. Since this there is no possible way for a spy to reach p_0, e_v is not a spy event. $\square$

6.5 Integrity Proofs in DR

The requester guarantees the integrity of the message it will receive, by adding in the first layer, a fresh name m, encrypted with a new public key $Pub(s)$. This value should be kept inside the message in order to be recognised. Since the name m is included in the message in the same way as p, and we have already proved the secrecy property for the response information p in section 6.4.2, we can state that m is kept as a secret along the protocol. In this case, we can ensure that nobody different from the requester has access to m. Since every helper must add some information to the message, and the only way to keep the m value is maintaining the already received contents, the helper must add its new data and cover the whole message with a new encryption layer generated with $Pub(s)$. Then, if it can be guaranteed that the name m persists in the message, and this nonce is never leaked (already verified), the integrity of the message, is never harmed.

This integrity property is verified by presenting a property which states that every message intended for the requester has the same structure which indicates that the nonce m is always present, and as we said, if m is kept as a secret, the integrity of the message is ensured. In order to verify this property, each output event occurring in the protocol must be verified, to ensure that there is no message intended for the requester, which appear without nonce m.

6.6 Integrity Property for the messages intended for the Requester

This theorem states that the same fresh name m will always appear in the same message identified with a request id Rid. This property among with the secrecy property for value m will ensure the integrity of the message.

Theorem 4. *Given a run of DR, $X_0 \in Peers(G)$, $Priv(k_0) \not\sqsubseteq t_0$, and the run contains an $Alice$ event a_1 labelled with action*

$$act(a_1) = Alice : (X_0) : i_0 : i : out\{Rid_0, Pub(k_0), \{m_0\}_{Pub(k_0)}\}_{Key(X,i)}$$

Where i_0 is a session index, i is an index which belongs to the set $f(X)$, Rid_0, m_0 and k_0 are names and $Pub(k_0)$ is a public key associated to the name k_0, then at every stage w the integrity of the message will be maintained.

Proof. We show the formalised proof in the following property:

$$Q(p,s,t,m_0) \Leftrightarrow \forall M \in \sigma(Rid_0,t) \,.\, M \sqsubseteq \{n_0, Rid_0, Pub(k_0), \psi\langle m_0\rangle[m_0/m]\}_{Key(X,Y)}$$

If we can show that at every stage w of the run $Q(p_w, s_w, t_w, m_0)$ then clearly the integrity of the message is maintained along all stages w in the run. Suppose the contrary, suppose that at some stage in the run, property Q does not hold, by freshness clearly $Q(DR, s_0, t_0, m_0)$. Let v by well foundedness be the first stage in th run st $\neg Q(p_v, s_v, t_v, m_0)$. From the freshness principle it follows

$$a_1 \longrightarrow e_v$$

and from the token game of nets $\{Rid_0, Pub(k_0), \{m_0\}_{Pub(k_0)}\}_{Key(X,i)} \in \sigma(Rid_0, t_{v-1})$ (Because messages are persistent in the net). The event e_v is an event in

$$Ev(DR) \equiv Alice : Ev(P_{Alice}) \cup Bob : Ev(P_{Bob}) \cup Spy : Ev(P_{Spy})$$

and from the token game of nets with persistent conditions is st

$$\sigma(Rid_0, e_v^o - e_{v-1}^o) \not\sqsubseteq \{n_0, Rid_0, Pub(k_0), \psi\langle m_0\rangle[m_0/m]\}_{Key(X,Y)} \wedge \forall m_i \in \sigma(Rid_0, e_v^o - e_{v-1}^o)\,,\, m_0 \gg m_i \quad (8)$$

Clearly e_v can only be an output event since $e_v^o - e_{v-1}^o = \emptyset$ for all input events e. Examining the output events of $Ev(DR)$, we conclude that $e_v \notin Ev(DR)$ reaching a contradiction.

Since we are analysing the integrity of messages intended for the requester, we will take a look at specific output processes where a particular message identified by a Request id Rid_0 occurs. (Where $Rid = Rid_0$). We explore these events in order to verify that the event e_v is different to all them.

Alice output events.

$$act(e_v) = Alice : (X) : j : i : out\, new\, (Rid, k, m)\{Rid, Pub(k), \{m\}_{Pub(k)}\}_{Key(X,i)}$$

Where $X \in Peers(G)$ and so $X \in s_0$, where Rid, m and k are names, $Pub(k)$ is a public key associated to the name k, j is a session index and i is an index which belongs to the set $f(X)$ where $i \in Peers(G)$ and so $i \in s_0$. Property 8 and the definition of message surroundings imply that $m_0 \gg \{Rid, Pub(k), \{m\}_{Pub(k)}\}_{Key(X,i)}$. Since $Rid = Rid_0$ then $m_0 \neq Rid$. And from the freshness property $m_0 \neq Pub(k)$. Then, if $m_0 = m$ then we reach a contradiction to property 8 because from the output principle it follows that $e_v^o - e_{v-1}^o = \{Rid_0, Pub(k_0), \psi\langle m_0\rangle[m_0/m]\}_{Key(X,i)}$. Therefore e_v can not be an A event with the above action.

Bob output events.

Case Fwd First output event

$$act(e_v) = Bob : (X) : j' : j : out\, \{Rid, Pub(k), \{\psi, p\}_{Pub(k)}\}_{Key(X,j)}$$

Where $X \in Peers(G)$ and so $X \in s_0$, where Rid, m and k are names, $Pub(k)$ is a public key associated to the name k, $p \in info$, j' is a session index and j is an index which belongs to the set $f(X)$ where $j \in Peers(G)$ and so $j \in s_0$.Property 8 and the definition of message surroundings imply that $m_0 \gg \{Rid, Pub(k), \{\psi, p\}_{Pub(k)}\}_{Key(X,j)}$. $Rid = Rid_0$ then $m_0 \neq Rid$. Since $p \in Info$ and so $p \in s_0$ from the freshness principle it follows that $m_0 \neq p$. If $m_0 = m$ then we reach a contradiction to property 8 because from the output principle it follows that $e_v^o - e_{v-1}^o = \{Rid_0, Pub(k_0), \psi\langle m_0\rangle[m_0/m]\}_{Key(X,j)}$. Then since property 8 must hold, $m_0 = Pub(k)$. By control precedence there exists an event e_u in the run st.

$$e_u \longrightarrow e_v$$

and

$$act(e_u) = Bob : (X) : j' : in\{Rid, m_0, \psi\}_{Key(Y,X)}$$

By the token game

$$\{Rid, m_0, \psi\}_{Key(Y,X)} \in t_{u-1}$$

where $n_0 \neq Pub(k_0)$ and so $\neg Q(p_{u-1}, s_{u-1}, t_{u-1})$ which is a contradiction because $u < v$.

Case Fwd Second output event

$$act(e_v) = Bob : (X) : j' : out\, \{n, Rid, Pub(k), \psi\}_{Key(X,Y)}$$

Where $X \in Peers(G)$ and so $X \in s_0$, where n, Rid, m and k are names, $Pub(k)$ is a public key associated to the name k and j' is a session index. Property 8 and the definition of message surroundings imply that $m_0 \gg \{n, Rid, Pub(k), \psi\}_{Key(X,Y)}$. $Rid = Rid_0$ then $m_0 \neq Rid$. Since $p \in Info$ and so $p \in s_0$ from the freshness principle it follows that $m_0 \neq p$. If $m_0 = m$ then we reach a contradiction to property 8 because from the output principle it follows that $e^o_v - e^o_{v-1} = \{n_0, Rid_0, Pub(k_0), \psi\langle m_0\rangle[m_0/m]\}_{Key(X,Y)}$. Property 8 must hold then, $m_0 = n$ or $m_0 = Pub(k)$. By control precedence there exists an event e_u in the run st.

$$e_u \longrightarrow e_v$$

and

$$act(e_u) = Bob : (X) : j' : in\{n, Rid, Pub(k), \psi\}_{Key(j,X)}$$

By the token game

$$\{n, Rid, Pub(k), \psi\}_{Key(j,X)} \in t_{u-1}$$

and $\neg Q(p_{u-1}, s_{u-1}, t_{u-1})$ since $\{m_0, Rid, Pub(k), \psi\}_{Key(j,X)} \in \sigma(m_0, t_{u-1})$ or $\{n, Rid, m_0, \psi\}_{Key(j,X)} \in \sigma(m_0, t_{u-1})$ and then $\sigma(m_0, t_{u-1}) \not\sqsubseteq \{n_0, Rid_0, Pub(k_0), \psi[m_0/m]\}_{Key(X,Y)}$, a contradiction follows because $u < v$.

Case $Triumph$ **output event**

$$act(e_v) = Bob : (X) : j' : out\,new(n)\,\{n, Rid, Pub(k), \{\psi, p\}_{Pub(k)}\}_{Key(X,Y)}$$

Where $X \in Peers(G)$ and so $X \in s_0$, where n, Rid, m and k are names, $Pub(k)$ is a public key associated to the name k, $p \in info$ and j' is a session index. Property 8 and the definition of message surroundings imply that $m_0 \gg \{n, Rid, Pub(k), \{\psi, p\}_{Pub(k)}\}_{Key(X,Y)}$. $Rid = Rid_0$ then $m_0 \neq Rid$. From the freshness principle, $m_0 \neq n$. Since $p \in Info$ and so $p \in s_0$ from the freshness principle it follows that $m_0 \neq p$. If $m_0 = m$ then we reach a contradiction to property 8 because from the output principle it follows that $e^o_v - e^o_{v-1} = \{n_0, Rid_0, Pub(k_0), \psi\langle m_0\rangle[m_0/m]\}_{Key(X,Y)}$. Then since property 8 must hold, $m_0 = n$ or $m_0 = Pub(k)$. By control precedence there exists an event e_u in the run st

$$e_u \longrightarrow e_v$$

and

$$act(e_u) = Bob : (X) : j' : in\{n, Rid, Pub(k), \psi\}_{Key(Y,X)}$$

By the token game

$$\{n, Rid, Pub(k), \psi\}_{Key(Y,X)} \in t_{u-1}$$

and $\neg Q(p_{u-1}, s_{u-1}, t_{u-1})$ since $\{m_0, Rid, Pub(k), \psi\}_{Key(Y,X)} \in \sigma(m_0, t_{u-1})$ or $\{n, Rid, m_0, \psi\}_{Key(Y,X)} \in \sigma(m_0, t_{u-1})$ and then $\sigma(m_0, t_{u-1}) \not\sqsubseteq \{n_0, Rid_0, Pub(k_0), \psi\langle m_0\rangle[m_0/m]\}_{Key(X,Y)}$, a contradiction follows because $u < v$.

Spy output events Since we have proved before that the private key $Priv(k)$ is never leaked, we can guarantee that no Spy can ever change the contents of the messages, then e_v is not a Spy event. □

7 Discussion

This paper presents two main ideas we want to extend, the first relies on the modelling and specification of a new set of constructions closely related to concurrency models. Although these are not new ideas and are present in other process calculi such as [MPW89, AG97, AF01, Hoa83, Car99], a pure inclusion of these kind of tools in SPL presents serious difficulties according to the inherent model of persistent networks. Therefore, by using the nominality of this calculus together with strong encryption mechanisms, this kind of constructions can be emulated without any intrusive changes to SPL operational semantics. Hence, providing a set of encodings allows a clean and straightforward translation between a broader subset of protocols models in different concurrency models mentioned before and SPL. However, a strong relation concerning the expressiveness is necessary to achieve a complete translation within them. Previous works establishing strong relations between lambda-calculus[Chu51] and the π calculus, and between persistent and non-persistent languages are presented by means of encodings[SW01, GSV04, PSVV04]. Relying on this concepts, an interesting strand of research could involve an encoding from SPL to the asynchronous π calculus in such a way that every calculus π-reducible can be translated to SPL in order to use its simple but powerful reasoning techniques.

Our second contribution we want to stand out relates to the formalisation and proof of new security properties using a process calculi. In particular, we have considered Integrity as one of the essential properties in order to guarantee the security of the system, particularly in applications where mobility involves extensibility of services, resources or functionality. Security information technologies have presented different approaches to tackle it, involving every one of the levels in information security, from ACID control mechanisms [SK86], to security protocols [ZS00] and policies [BF03]. However, reasoning techniques provided by process calculi, in particular SPL, brings the necessary flexibility to construct a powerful framework to prove different security properties, a clear advantage from specific-driven models.

References

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 104–115, New York, NY, USA, 2001. ACM Press.

[AG97] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus, 1997.

[BF03] Stefano Bistarelli and S. Foley. Analysis of integrity policies using soft constraints. In *Proceedings of IEEE Workshop Policies for Distributed Systems and Networks*, pages 77–80, 2003.

[BMWZ05] Matthias Bender, Sebastian Michel, Gerhard Weikum, and Christian Zimmer. The minerva project: Database selection in the context of p2p search. In *BTW*, pages 125–144, 2005.

[BS04] S. Baset and H. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. Technical report, Computer Science Department,Columbia University, September 2004.

[Car99] Luca Cardelli. Mobilility and security, 1999.

[CCM02] Mario José Cáccamo, Federico Crazzolara, and Giuseppe Milicia. The ISO 5-pass authentication in χ-Spaces. In Youngsong Mun and Hamid R. Arabnia, editors, *Proceedings of the Security and Management Conference (SAM'02)*, pages 490–495, Las Vegas, Nevada, USA, June 2002. CSREA Press.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with r̈eadersänd ẅriters̈. *Commun. ACM*, 14(10):667–668, 1971.

[Chu51] A. Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951. (second printing, first appeared 1941).

[Cra03] Federico Crazzolara. Language, semantics, and methods for security protocols. Doctoral Dissertation DS-03-4, brics, daimi, May 2003. PhD thesis. xii+160.

[CW01] Federico Crazzolara and Glynn Winskel. Events in security protocols. In *ACM Conference on Computer and Communications Security*, pages 96–105, 2001.

[DGOR04] Juan Francisco Díaz, Gustavo Gutierrez, Carlos Alberto Olarte, and Camilo Rueda. Cre2: A cp application for reconfiguring a power distribution network for power losses reduction. In *CP*, pages 813–814, 2004.

[Ese02] A. Esenther. Instant co-browsing: Lightweight real-time collaborative web browsing, 2002.

[GK03] Nathaniel S. Good and Aaron Krekelberg. Usability and privacy: a study of kazaa p2p file-sharing. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 137–144, New York, NY, USA, 2003. ACM Press.

[GRS99] D. Goldschlag, M. Reed, and P. Syverson. Onion routing for anonymous and private internet connections. *Communications of the ACM (USA)*, 42(2):39–41, 1999.

[GSV04] Pablo Giambiagi, Gerardo Schneider, and Frank D. Valencia. On the expressiveness of infinite behavior and name scoping in process calculi. In Igor Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 2004.

[Hoa83] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 26(1):100–106, 1983.

[Mil99] Robin Milner. *Communicating and Mobile systems. The Pi Calculus*. Cambridge University Press, 1999.

[Mil02] Giuseppe Milicia. χ-Spaces: Programming Security Protocols. In *Proceedings of the 14th Nordic Workshop on Programming Theory (NWPT'02)*, November 2002.

[MKL$^+$02] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing. Technical Report HPL-2002-57, HP Labs, March 2002.

[MPW89] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. Technical Report -86, 1989.

[Per96] Charles Perkins. IP Mobility Support - RFC2002. IETF RFC Publication, 1996.

[PR99] N. De Palma and B. Riveill. Dynamic reconfiguration of agent-based applications, 1999.

[PSVV04] Catuscia Palamidessi, Vijay Saraswat, Frank D. Valencia, and Bjorn Victor. Linearity and persistence in the pi-calculus. unpublished, 2004.

[PV01] Catuscia Palamidessi and Frank Valencia. A temporal concurrent constraint programming calculus. In Toby Walsh, editor, *Proc. of the 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239, pages 302–316. LNCS, Springer-Verlag, 2001.

[Rip01] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network, 2001.

[SK86] Abraham Silberschatz and Henry F. Korth. *Database System Concepts, 1st Edition*. McGraw-Hill Book Company, 1986.

[SW01] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[VS05] Geoffrey M. Voelker and Scott Shenker, editors. *Peer-to-Peer Systems III, Third International Workshop, IPTPS 2004, La Jolla, CA, USA, February 26-27, 2004, Revised Selected Papers*, volume 3279 of *Lecture Notes in Computer Science*. Springer, 2005.

[WHY$^+$04a] Helen J. Wang, Yih-Chun Hu, Chun Yuan, Zheng Zhang, and Yi-Min Wang. Friends troubleshooting network: Towards privacy-preserving, automatic troubleshooting. In Voelker and Shenker [VS05], pages 184–194.

[WHY$^+$04b] Helen J. Wang, Yih-Chun Hu, Chun Yuan, Zheng Zhang, and Yi-Min Wang. Friends troubleshooting network: Towards privacy-preserving, automatic troubleshooting. In Voelker and Shenker [VS05], pages 184–194.

[ZS00] Yongguang Zhang and Bikramjit Singh. A Multi-Layer IPsec protocol. In *USENIX 2000*, pages 213–228, 2000.