Dynamic Verification and Control of Mobile Peer-to-Peer Systems

George Spanoudakis, Christos Kloukinas and Kelly Androutsopoulos
Department of Computing,
City University,
London,
United Kingdom,
{G.Spanoudakis, C.Kloukinas}@soi.city.ac.uk, kellyandrou@googlemail.com

Abstract

The development of dependable mobile P2P systems is an inherently challenging task since such systems may operate in largely uncontrolled environments and may engage new peers or lose existing ones without any form of centralised control. In these circumstances, dependability and security can be enhanced through the runtime monitoring (a.k.a. dynamic verification) of the compliance of the system behaviour against specific dependability and security properties and the execution of control in cases where properties are violated. In this paper we present a framework for the dynamic verification and control of mobile P2P systems, which uses peer-specific monitoring policies to specify application-level properties. The deployment of this framework for monitoring system behaviour adds an extra layer of security and dependability checking, which is independent from checks performed directly by the P2P system that is being monitored. Thus, it makes the system more fault-tolerant and enables event logging that could be used for further analysis and prevention of attacks.

1. Introduction

Starting from the same types of peer-to-peer (P2P) applications that have been dominant in desktop platforms (e.g. instant messaging, voice over IP and file sharing applications), mobile P2P systems are increasingly covering a wide spectrum of functionalities such as B2C and C2C e-commerce applications, and mobile game engines. This tendency is due to some key benefits of the decentralisation of P2P architectures over their centralised counterparts, including reduced vulnerability to single-point failures and higher scalability [1]. The emergence of mobile P2P applications has been enabled by the proliferation of mobile devices equipped with technologies such as WiFi and GPRS.

However, despite the benefits that arise from the decentralised and dynamic nature of mobile P2P systems, these very characteristics are also posing some significant challenges for their dependability and security. As individual peers may enter or leave a mobile P2P system at will and without notice, the availability of peers and the services that they offer cannot be taken for granted. Peer availability may also be affected by the limited power and computational capacity of mobile devices, the availability of connectivity in different locations, and the usage patterns of individual mobile peers.

In such circumstances, ensuring that P2P applications have been developed using sound software engineering practices and incorporate a set of basic security mechanisms is not sufficient for guaranteeing security and dependability. Therefore, the preservation of these properties by mobile P2P systems needs to be dynamically verified through monitoring at runtime. Dynamic verification complements system testing and static verification, as none of the latter techniques can guarantee the correctness of system behaviour at runtime, either because it is difficult to foresee and check all the possible conditions that may arise in the real operational context of a system (system testing) or ensure that the models used for static verification are preserved by system implementations.

Several dynamic verification techniques have been developed (e.g. [2][8][11][13][21]), providing mechanisms to capture events from software systems at runtime and check whether these events satisfy specific properties, which are typically expressed in some temporal logic language. However, as they focus mainly on computing platforms where resource scarcity is not a significant constraint and systems that are not as dynamic and uncontrollable as mobile P2P systems, these techniques do not address adequately some issues

which are essential for the dynamic verification of mobile P2P systems, namely:

- the dynamic negotiation between mobile peers at run-time in order to enable the activation of monitoring activities,
- the need to have a monitoring service that is not deployed on the same machine as the peers that it monitors, in order not to drain the computational and power resources of mobile devices,
- the secure emission of events from mobile P2P systems required for monitoring, and
- the dynamic execution of actions to prevent or rectify detected violations of the monitored properties during the execution of peer applications.

In this paper, we present a dynamic verification framework for mobile P2P systems (shortly referred to as "DVF" in the rest of the paper) that addresses these issues and constitutes the first such framework which, to the best of our knowledge, is available on a mobile computing platform. The use of DVF for monitoring and controlling mobile P2P systems behaviour adds an extra layer of checking security and dependability that is independent from any checks performed by the peers themselves. Thus, DVF makes mobile P2P systems more fault-tolerant and enables logging of events which could be used for further diagnostic analysis and prevention of future attacks.

Central to the approach that underpins DVF is the use of *peer-specific monitoring policies*. These policies specify *application level properties* that need to be monitored, concerning a peer itself or its collaborators in a network, using a language based on Event Calculus [19], the *actions* that should be taken when the monitored properties are violated, and the *permissions* that a peer gives to its collaborators regarding the monitoring and control of its own activities. Furthermore, the DVF supports the negotiation and activation of monitoring policies between mobile peers and the collection and transmission of events from mobile peers for the monitoring of these policies. It also offers control capabilities that enable reactions to identified violations of the monitored properties.

The rest of this paper is structured as follows. In Section 2, we present the architecture of DVF. In Section 3 we introduce the language for specifying DVF policies. In Sections 4 and 5, we present the negotiation and control capabilities of DVF. In Section 6, we give an overview of the implementation of DVF and in Section 7 we discuss related work. Finally, in Section 8, we conclude and outline our plans for future work.

2. Architecture of the Dynamic Verification Framework

The main characteristic of DVF is that it decouples monitoring from event capturing and control, assigning responsibility for the latter two activities to individual peers and responsibility for the former activity to external monitors. Furthermore, the verification framework deploys an event notification infrastructure, supporting the transmission of events from peers to the monitors and the results of the monitoring process in the opposite direction. As shown in Figure 1, the DVF consists of three basic components: *Monitoring-Enabled Peers* (MEPs), *monitors* and *event brokers* (EBr). These components are described below.

2.1 Monitoring enabled peers

Monitoring enabled peers are peers that incorporate a *peer verification controller* (PVC). A PVC collects events during the operation of a peer and publishes them to event brokers, so that they can be distributed to appropriate monitors. A PVC has also responsibility for receiving notifications of the results of the monitoring process and taking control actions on the individual peer as required by these results (e.g., dropping messages exchanged between a peer and its collaborators). As shown in Figure 2, a PVC is provided as part of the basic runtime infrastructure that enables the formation of peer networks (e.g. peer registration, authentication and discovery) and the communication between the individual peers in them. Thus, when a peer application is built using this infrastructure, it automatically incorporates a PVC. A PVC internally consists of a *controller*, a *policy parser* and a *negotiation manager*.

The PVC controller intercepts all the incoming and outgoing messages, which are exchanged between its host peer and other peers, and publishes these messages in the form of encrypted events to the event broker of the DVF, according to its *event exposition rules*. These are determined dynamically through the *monitoring policy* of the host peer itself and agreements that it may have made with other peers regarding the exposition of its own events. After sending a message to an event broker, the controller may block it until it receives a notification that the message does not violate any rule or permit its transmission and wait for the asynchronous notification of monitoring results. Subsequently, when it receives the monitoring results that relate to the message, the PVC controller applies the actions required by the active monitoring policies.

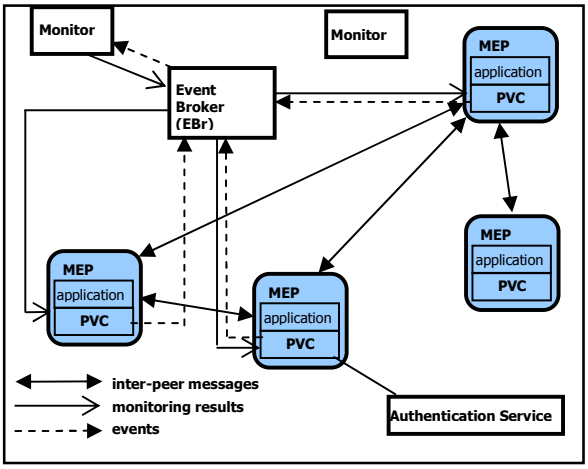

Figure 1. Architecture of DVF

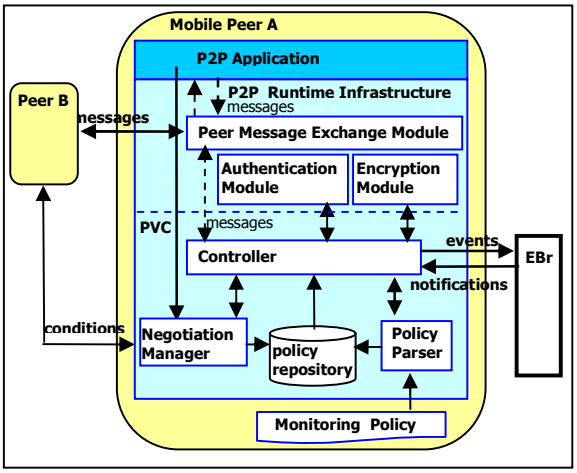

Figure 2. Peer verification controllers

The *policy parser* of the PVC is responsible for parsing the monitoring policy of the host peer and creating a repository with information about the types of events that should be intercepted, the properties against which the intercepted events should be checked and the actions that should be taken if the properties are violated (see Section 3). The policy also includes information about the events that the PVC may expose to other peers and the events that it should request from these peers in return.

Finally, the *negotiation manager* is the component of the PVC that enables it to negotiate with external peers for the reception and exposition of events that are necessary for checking the active monitoring policies at each side. The negotiation process is driven by the monitoring policies of the involved peers as we explain in sections 3 and 4. It should also be noted that the PVC comes pre-assembled with the peer communication infrastructure and signed off so that it cannot be circumvented or tampered with.

2.2 Monitors

The monitors are the components of DVF that carry out property checks by analysing the peer events at runtime. The DVF may employ more than one monitor, each having responsibility for different nodes of a peer network. A monitor is appointed to check specific properties by the PVC of an individual peer and can reside either on the peer itself or on an external device. The latter possibility is necessary for the provision of monitoring services to peers running on devices that cannot themselves support monitoring due to their limited resources, such as smart phones and PDAs.

Following its appointment by the PVC of a peer, a monitor subscribes to an event broker of DVF, in order to receive the events needed for checking the properties assigned to it and notifies property violations to the event broker so that the interested PVCs will be informed about them.

The monitor is a reasoning engine that checks whether the Event Calculus formulae that specify the required properties inside a monitoring policy are satisfied by the events which are generated by the peer PVCs at runtime and other events that can be derived from them. The derivation of events is based on deductive reasoning, performed by the reasoning engine itself. A detailed account of the algorithms that underpin the operation of monitors is beyond the scope of this paper and may be found in [16].

2.3 Event brokers

The event broker (EBr) in DVF offers the infrastructure needed for transmitting events from peers to monitors and monitoring results from monitors to peers. The event broker manages the subscriptions to the “channels” between publishers of messages and their respective subscribers. The use of the publish-subscribe event reporting infrastructure in DVF keeps the verification framework separate from the actual P2P service, allowing the latter to operate independently and adopt any overlay topology and mode of service provision that it needs. Thus, the P2P service developer is given freedom to design the service, without having to take into account the specifics of the monitoring framework which operates in its own overlay. At the same time, the monitoring framework is not “tied” to the service, allowing various different services to be monitored, without modifying the system to suit each one separately. Furthermore, the

use of the publish-subscribe architecture allows DVF to work well in an environment where peers come and go quickly and unpredictably, and does not overload the peer communication infrastructure with message transmissions required for monitoring.

The security of communications in publish-subscribe architectures can be ensured as demonstrated in [18][22]. To preserve confidentiality in DVF, the EBr manipulates encrypted publications, without having access to their actual contents. This is achieved through the use of secret tokens acting as aliases to the actual information exchanged, giving the EBr enough information to manage subscription and publication messages without knowing what a token refers to (see Section 4 for more details).

```
Policy policy_name
  [Rule RuleID String RuleFormula <formula>
    Assumptions
     [AssumptionID String
      AssumptionFormula <formula>]*
    AppliesTo <peer-list-type>*
    [Action <action-type>]*
  ]+
  [EventExposition  <event_exposition_type>]*
  Timeout Duration [drop|forward]
  Lifetime [until Date | permanent]
```

Figure 3. Monitoring policy specification language

3. Specification of monitoring policies

The operation of a PVC at runtime is driven by the monitoring policy of its host peer. Policies are specified according to the language shown in Figure 3 (an XML implementation of this language is described in [15]). A policy contains: (i) one or more *rules* specifying the properties that should be monitored at runtime, (ii) the types of events that the PVC is allowed to expose to other peers at runtime for their monitoring needs if requested (*EventExposition* elements), (iii) a *timeout* value determining the maximum time that an event can be blocked whilst a PVC waits for monitoring results, and (iv) a *lifetime* value which determines for how long the policy will be valid.

3.1 Specification of policy rules

The specification of a rule in a monitoring policy consists of an identifier that uniquely identifies the rule within the policy (*RuleID*), a formula that defines the logical form of the rule (*RuleFormula*), the peers that the rule *applies to* (i.e., the peers against which the rule should be monitored), a set of *assumptions*, and a set of *actions* that define the ways in which *DVF* should react when the rule is violated.

Formulae are defined in Event Calculus (EC [19]), a first-order temporal logic language which can be used for representing and reasoning about *events* and their effects over time. An event in EC is an occurrence that takes place at a specific instance of time (e.g., invocation of a system operation, reception or dispatch of a message) and may have an effect. The effects of events in EC are represented by *fluents*, i.e., conditions which may change over time. A fluent may, for example, specify a condition indicating that a peer has received a message or that following the receipt of a message an internal variable of a peer has been set to a specific value. EC fluents are initiated or terminated by an event. Event occurrences are represented by the predicate $Happens(e,t,\Re(t_1,t_2))$. This predicate denotes that the instantaneous event *e* occurs at a time point *t* within the time range $\Re(t_1,t_2)$. The range boundaries can be specified by either time constants or arithmetic expressions over the time variables of other predicates. The initiation or termination of a fluent *f* due to the occurrence of an event *e* at time *t* is denoted by the predicates *Initiates(e,f,t)* and *Terminates(e,f,t)*, respectively. Two additional predicates, namely *Initially(f)* and *HoldsAt(f,t)* may also be used to denote that a fluent *f* holds at the start of the execution of a system and that *f* holds at time *t*, respectively.

Fluents use the form $relation(Object_1, \ldots, Object_n)$ and events are restricted in our EC-based policy language to exchanges of messages between peers. A message can invoke an operation in a peer or return results following the execution of an operation. Events are specified using the following generic form:

event(_id, _sender, _receiver, _sig, _source)

In this form:

- *_id* is the unique identifier of the event
- *_sender* is the identifier of the peer that sends the message.
- *_receiver* is the identifier of the peer that receives the message.
- *_sig* is the signature of the operation or the type of the message that the event refers to.
- *_source* is the identifier of the peer from which the event was captured.

Examples of rules specified according to the policy language of DVF are shown in Figure 4. For example, *Rule_1* should be monitored against the peer that owns the policy, as the keyword *self* in its *AppliesTo* clause denotes. The rule checks whether after the peer has sent a message of type *authorise*, requesting the

authorisation of *_i* by an external peer *_B*, it will receive a message of type *authorisation*, signifying the authorisation of *_i* from *_B*, within t_u time units. Essentially, *Rule_1* monitors the (bounded) availability of peer *_B* which provides authorisation services and the communication channel between it and the host peer.

Assumptions in policies express how the state of a P2P system is affected by the events that occur during its operation. These effects are expressed by formulae that indicate the initialisation and termination of fluents by different events. Whilst monitoring the rules, the monitor uses the assumptions to deduce the status of fluents, i.e. whether or not a fluent holds at a particular instance of time. For example, *Rule_2* of Figure 4 checks cases where an external peer (i.e., peer *_A* in the rule) sends a *request* message to the current peer for a piece of information *_i* (e.g., a file). In such cases, the rule demands that the external peer *_A* must have been authenticated at the recipient before the request is received. The authentication of *_A* at *_self* is denoted by *HoldsAt(authenticated(_A, _self), t)*. *Rule_2*'s assumption *A1* is used at runtime to establish whether the fluent *authenticated(_A, _self)* holds at the time of the receipt of the request from *_B* or, equivalently, if *_A* has been authenticated at *_self* at this time. According to *A1*, the fluent *authenticated(_A, _self)* is initiated (i.e., becomes true) when a message confirming the authenticity of *_A* is received by the peer *_self*, following a request for the authentication of _A to an external authority (peer) _B. The authentication request is represented by the event *e(_eID1, self, _B, authenticate(_A), _self)* in the formula A1 and the response to it by the event *e(_eID2, _B, self, authentication(_A), _self)*.

During monitoring, the fluent *authenticated(_A, _self))* is obtained by deduction from *A1* and the EC axioms. More specifically, when the events that signify the dispatch of the authentication request message and the response to it occur, the predicate *Initiates(e(_eID2, _B, _self, authentication(_A),_self), authenticated(_A,self), t2))* is derived from A1 by deduction. After the initialisation of this fluent, the predicate *HoldsAt(authenticated(_A, self), t)* can be derived using an axiom of EC which states that a fluent will hold at any time point after it is initiated, unless there has been an event terminating it in between.

3.2 Specification of actions

DVF policies also specify the control actions that should be executed by the PVC, following rule violations. These actions can be of three different types: drop actions, violation notification actions, and negotiation actions.

Policy policy-1
Rule RuleID Rule_1
RuleFormula
Happens(e(_eID1, _self, _B, authorise(_i), _self), t1, R(t1,t1))
⇒Happens(e(_eID2, _B, _self, authorisation(_i), _self),t2,
R(t1,t1+tu)))
AppliesTo _self
Action notify(_eID1, _self, _B)

Rule RuleID Rule_2
RuleFormula
Happens(e(_eID1, _A, _self, request(_i), _self), t , R(t,t))
⇒ HoldsAt(authenticated(_A, self), t)
Assumptions
AssumptionID A1
AssumptionFormula
Happens(e(_eID2, _self, _B, authenticate(_A), _self), t1,
R(t1,t1))
∧ Happens(e(_eID3, _B, _self, authentication(_A), _self), t2,
R(t1,t2))
⇒ Initiates(e(_eID3, _B, _self, authentication(_A), _self),
authenticated(_A,self), t2))
AppliesTo _self, Peer-role-A
Actions drop(_eID1, _self)

Rule RuleID Rule_3
RuleFormula
Happens(e(_eID1, _A, _self, request(_i), _self), t1, R(t1,t1))
∧ ¬∃t1. Happens(e(_eID2, _A, _self, request(_i), _self), t2,
R(t2,t1))
⇒ HoldsAt(negotiated(_self,_A), t)
Assumptions
AssumptionID A1
AssumptionFormula
Happens(e(_eID1, _self, _self, start_neg(_A), _self), t1,
R(t1,t1))
∧ Happens(e(_eID2, _A, _self, confirm_neg(_A), _self), t2,
R(t1,t2))
⇒ Initiates(e(_eID2, _A, _self, confirm_neg(_A),_self),
negotiated(_self,_A), t2))
AppliesTo self
Actions negotiate(_eID1, _A)

Rule RuleID Rule_4
RuleFormula
Happens(e(_eID1, _self, _A, dispatch(_i), _self), t1, R(t1,t1))
⇒ HoldsAt(negotiated(_self,_A), t)
Assumptions {}
AppliesTo self
Actions drop(_eID1)

EventExposition
Timeout 1000 drop
Lifetime permanent

Figure 4. Policy example

Drop actions prevent the dispatch of the peer message that has caused the violation of a rule to the peer that is the intended recipient of the message and is specified as *drop(eventID, peerID$_1$, …, peerID$_n$)*, where *eventID* is the identifier of the event that is involved in the violation of the rule, and *peerID$_1$*, …, *peerID$_n$* the peers to be notified of the dropped message.

Violation notification actions can be taken in cases where the event that has caused the violation should not be dropped but a notification of the violation should be send to certain peers and are specified as *notify(eventID, peerID$_1$, …, peerID$_n$)*.

Finally, negotiation actions can be taken in cases where the violation of a rule by an event should trigger the negotiation process between peers and are specified as *negotiate(eventID, peerID$_1$)*, where *peerID* is the identifier of the peer with whom the negotiation must take place.

As Figure 4 shows, in the case of *Rule_1* the action that should be taken is *notify(_eID1, self, _B)*. This action causes the PVC of the peer _self to send a notification message to the peers _self and _B, to inform them that the event _eID1 has violated *Rule_1*. When violations of *Rule_2* occur, the drop action *drop(_eID2, _self)* should be executed. This will result in the notification of the violation of *Rule_2* to _self but as the drop action specifies no other recipients of the notification, the peer _A, which had requested item _i causing the violation, will not be informed of the drop of the message. Of course, the absence of notifications should be used with care, as peers may end up waiting for a response indefinitely.

4. Negotiation

A peer MEP$_1$ starts the negotiation process with another peer MEP$_2$ when it needs events of MEP$_2$ to monitor its own policy. The negotiation process between two peers can also be triggered by forcing the PVC of a peer to take a negotiation action, following the violation of a rule in its monitoring policy. Such a rule will typically require that when MEP$_1$ receives some particular message from another peer MEP$_2$, it should negotiate with MEP$_2$ the exposition of events from it unless it has already done so.

For example, if MEP$_1$ has the policy specified in Figure 4, then according to *Rule_3* when it receives a *request(_i)* message from a peer MEP$_2$, the fluent *negotiated(MEP$_1$, MEP$_2$)* must hold, indicating that a successful negotiation between MEP$_1$ and MEP$_2$ has already taken place. If that is not the case, *Rule_3* will be violated and the PVC of MEP$_1$ will execute the action *negotiate(_eID1, MEP$_2$)* as specified by the rule. This action will trigger the negotiation process between the PVCs of MEP$_1$ and MEP$_2$.

At the start of the negotiation process, the PVC of MEP$_1$ will identify the rules that apply to the role of peer MEP$_2$ and from these rules it will subsequently identify the events that it will need from MEP$_2$ in order to check these rules and the actions that should be executed if the rules are violated. Using this information, the PVC of MEP$_1$ will construct a *condition list* of the following form (1) and send it to MEP$_2$ for approval.

```
(ev-type_i,                                          (1)
  ((rule_1, (action_11, …, action_1L)),
    …,
   (rule_n, (action_n1, …, action_nM))))
(i=1,…,k)
```

An element *i* in this list denotes the type of the events of MEP$_2$ that will be required (i.e. *ev-type$_i$)*, the rules against which events of this type will be checked (*rule$_1$* ,…., *rule$_n$*) and the actions that should be taken if one of these rules is violated (e.g., *action$_{11}$*, …, *action$_{1L}$* for *rule$_1$*). After receiving the condition list, the PVC of MEP$_2$ will check it against the event exposition list of its own policy. If the exposition list allows it to accept the conditions sent by MEP$_1$, it will update its internal active policy, so as to send the agreed events to MEP$_1$.

Continuing with our previous example, suppose that the role of MEP$_2$ is *Peer-role-A*. Based on this, MEP$_1$ will need to monitor whether the operation of MEP$_2$ is compliant with *Rule_2*. From this rule, it can then construct the following condition list and send it to MEP$_2$ for negotiation:

```
[(e(_eID1, _A, MEP2, request(_i), MEP2),
       (Rule_2, (drop(_eID1, MEP2))),
(e(_eID2, MEP2, _B, authenticate(_A), MEP2),
       (Rule_2, ()),
(e(_eID3, _B, MEP2, authentication(_A), MEP2),
       (Rule_2, ()))]
```

The three event types in the above list are extracted from *Rule_2* after replacing *_self* with MEP$_2$ as the latter peer will become the subject of the monitoring of *Rule_2* in this case. Assuming that the event exposition list in the monitoring policy of MEP$_2$ is

```
EventExposition
(request(_i),[peer-role-B],
[notify(request(_i), _self)])
       (authenticate(_X),[peer-role-B],[])
       (authentication(_X),[peer-role-B],[])
```

the latter peer will not accept the condition list of MEP$_1$, as the actions that should be applied for violations caused by *request(_i)* events are not included in its permissible actions for this type of events. Thus, the negotiation will fail. However, if the action *drop(_eID1, MEP2)* was in the permissible action list for *request(_i)* then the negotiation would have been completed successfully.

When the PVC of MEP$_1$ starts the negotiation process, it sends a message *start_neg(MEP$_2$)* to its peer to indicate this and, if the negotiation process is completed successfully, it sends the message *confirm_neg(MEP$_2$)*. These messages are also sent to the monitor, which from assumption *A1* of *Rule_3* will establish the fluent *negotiated(MEP$_1$, MEP$_2$)*. Thus, the next time that MEP$_2$ sends a *request(_i)* message to

MEP_1 the fluent will hold and there will be no need to start the negotiation process again.

After the conditions are accepted in the negotiation process, MEP_1 will need to establish two confidential communication channels that will allow the PVC of MEP_2 to send the events required for monitoring to the monitor of MEP_1 and the monitor to notify the results of the monitoring process back to MEP_1 and MEP_2. In DVF these communication channels are established through the event broker.

(1)	MEP_1 → MEP_2:	$T_e(i)$, $T_{mr}(i)$
(2)	MEP_1 → EBr:	adv(M, $T_{mr}(i)$, t)
	MEP_2 → EBr:	adv(MEP_2, $T_e(i)$, t)
(3)	MEP_1 ← MEP_2:	H($T_e(i)$, $T_{mr}(i)$)
(4)	MEP_2 → EBr:	sub(MEP_2, $T_{mr}(i)$, t)
	MEP_1 → EBr:	sub(MEP_1, $T_{mr}(i)$, t), sub(M, $T_e(i)$, t)
(5)	MEP_1,M ← EBr:	H(M, $T_{mr}(i)$, t), H(M, $T_e(i)$, t)
	MEP_1,MEP_2←EBr:	H(MEP_2, $T_e(i)$, t), H(MEP_2, $T_{mr}(i)$, t)
(6)	MEP_1 ← M:	H(H(M, $T_{mr}(i)$, t)), H(M, $T_e(i)$, t)))
	MEP_1 ← MEP_2:	H(H(MEP_2, $T_e(i)$, t), H(MEP_2, $T_{mr}(i)$, t))
(7)	MEP_1 → M:	$T_e(i)$, $T_{mr}(i)$, $seed_e(i)$, $seed_{mr}(i)$
(8)	MEP_1 ← M:	H(Key_e, Key_{mr})
(9)	MEP_1 → MEP_2:	$seed_e(i)$, $seed_{mr}(i)$
(10)	MEP_1 ← MEP_2:	H(Key_e, Key_{mr})
(11)	MEP_1 → MEP_2:	reportToEBr()

Figure 5. Protocol of establishing event and results notification channels[1]

DVF assumes that event brokers are not trusted entities and therefore they should be able to manage the subscriptions and publications of events and monitoring results without having access to their contents. To achieve this, the events and monitoring results are encrypted and the necessary keys for the decryption of this information are generated outside the event broker and are not made available to it. The event broker gets only tokens that identify the notification channels and enable it to distribute the encrypted messages to the appropriate subscribers. Tokens essentially provide aliases to the actual information exchanged, giving the event broker sufficient information for managing subscriptions and routing publications, without knowing what a token refers to or being able to deduce the actual type of the transmitted messages or other information from it. The protocol for creating the tokens and decryption keys and establishing the event and notification reporting channels is shown in Figure 5.

This protocol is implemented by the PVCs of the peers which are involved in the negotiation. In the following, we explain the protocol in reference to our previous example where the peer MEP_1 requested specific types of events for monitoring from MEP_2.

The execution of the protocol starts after MEP_2 agrees to provide the types of events requested by MEP_1. At this point MEP_1 will need to coordinate the process of creating the two necessary channels between MEP_2's PVC and the monitor M through EBr. Thus, after MEP_1 receives a notification of the acceptance of its condition list from MEP_2, it creates unique tokens to reference MEP_2's event channel ($T_e(i)$) and the monitoring results channel ($T_{mr}(i)$) and forwards them to MEP_2 (step (1)). It also sends an advertisement message to EBr, indicating that the monitor M will publish monitoring results referenced by the token $T_{mr}(i)$ (step (2)). MEP_2 also sends an advertisement message to EBr, indicating that it will publish events referenced by the token $T_e(i)$ (step (2)), and acknowledges the receipt of the tokens to MEP_1 (step (3))[2]. Following this, MEP_2 asks EBr to subscribe to the results that will be published by the monitor (step (4)). In parallel, after receiving the acknowledgement of the receipt of the tokens by MEP_2, MEP_1 sends a message to EBr to subscribe M to the events that will be published by MEP_2 and itself to the result channel of M. Following the acknowledgement of the created subscriptions from EBr (step (5)), and from MEP_2 and M (step (6)), MEP_1 forwards two seeds, which are necessary for the local creation of symmetric encryption/decryption session keys for the event and results channel, to MEP_2 and M (steps (7) and (9)). These keys are related to the tokens (and as such, to the specific events that have been negotiated) and, therefore, cannot be used to decrypt any other channels.

MEP_2 and M use the same symmetric key generation function as MEP_1 to generate Key_e and Key_{mr} from the two pairs of token-seeds ($T_e(i)$,$seed_e(i)$) and ($T_{mr}(i)$,$seed_{mr}(i)$), respectively. This function is provided by the basic runtime infrastructure within which the PVCs of MEP_1 and MEP_2 are embedded (see [11]) and is also implemented by the monitors of DVF. The process of establishing confidential channels is

[1] If two or more steps have the same number in the protocol, the order of their execution is not important.

[2] The acknowledgement messages sent during the execution of protocol contain the hash value of the parameters of the message they acknowledge and are denoted by H(message). This enables the verification of message integrity. The hash function H that is used is provided by the peer infrastructure that embeds PVC.

concluded successfully only when M and MEP_2 acknowledge to MEP_1 the key creation (steps (8) and (10)), through the hash value of the keys. MEP_1 matches the hash values of the acknowledgements with the hash value of the keys that it has created locally and it concludes that the process has completed successfully only if a match is found. At the end of this process, only MEP_1, MEP_2 and M possess the keys that can be used to encrypt and decrypt the contents of the event and monitoring results channels. Thus, EBr cannot read the contents of the "channels" corresponding to these tokens, despite knowing the tokens that will enable it to forward publications to subscribers. The channel establishment process is aborted if at any point MEP_1 does not receive the acknowledgements that it expects within a pre-specified time period. This protocol follows the process for creating cryptographic keys in the Secure Sockets Layer (SSL) protocol [10] and ensures that the event notification infrastructure is engaged at runtime in a flexible but secure manner.

```
EventGeneration(m: PeerMessage)
i_time = current_time()
Create an event e(m) for m
timeout = ActivePolicy(e(m)).timeout
timeleft = timeout.duration
e(m).D_Rules = {R| R ∈ ActivePolicy(e(m)).Rules
        ∧ ∃a. a ∈ ActivePolicy(e(m).Actions
        ∧ (a = drop-action) ∧ a.rule = R}
e(m).ND_Rules = ActivePolicy(e(m)).Rules − D_Rules
If e(m).D_Rules = ∅ then
  If e(m).ND_Rules ≠ ∅
    Send e(m) to Brokers(Type(e(m)).List)
  Send m to its destinations
  return
EndIf

Send e(m) to Brokers(Type(e(m)).List)
While e(m).NoViolation ∨ e(m).D_Rules ≠ ∅ Do
  //wait for notifications
  wt = i_time + timeleft – current_time()
  If (rcv(e(m).chan,NewN,wt) = Timeout) Then
    If (timeout.action = forward)
      send m to its destinations
    return
  EndIf
  R = NewN.rule
  If R ∈ e(m).D_Rules ∧ NewN.violation Then
      DA = DropActions(R)
      Apply(DA)
      e(m).NoViolation = False
  EndIf
EndWhile

If e(m).NoViolation
  send m to its destinations
End Controller
```

Figure 6. Event generation algorithm

5. Control

The application of control actions in DVF is the responsibility of the *PVC controller* and is driven by: (a) the active monitoring policy that exists in a peer and (b) the agreements that the peer may have made with external peers after negotiation (if any). As we discussed in Section 3, the monitoring policy of a peer may define some control actions for each of the rules to be monitored. Each of these actions must refer to a specific event within the rule that it applies to. In the policy of Figure 4, for instance, the drop action specified for *Rule_3* refers to the event *e(_eID2,_self,_A, request(_i), _self)* in the rule formula[3] and, therefore, it can be applied only to runtime events that match this event in the formula.

Based on the specification of the actions, the policy parser creates an action list for the different types of events that have been identified in a policy. The elements of this list have the same form as (1) indicating the rules against which events of a particular type should be checked and the actions that should be applied if a violation of these rules is caused by the events. This list constitutes the internal *active monitoring policy* of the peer and is updated through condition negotiations with other peers. The PVC implements control as specified by the algorithms *event generation* and *notification handling* which are shown in Figure 6 and Figure 7, respectively.

As shown in Figure 6, the PVC controller constructs a new event for each message it catches and finds the set of rules that need to be checked for the event and have at least one drop action defined for it (D_{Rules}). If this set is empty, then the controller transmits the event to the event broker without waiting for any monitoring results as these can be handled asynchronously. If, however, there are rules with drop actions, then the controller must ensure that all these rules have been satisfied before allowing the message to be transmitted to its destination. Thus, it waits to receive notifications (*NewN*) for this event. If a timeout occurs first and the timeout action of the policy is to forward the message, then this is done and the controller returns immediately. Otherwise, if a violation of a rule with a drop action for the event is notified, the controller drops the event and stops waiting for any further notifications of monitoring results for the event as these can again be handled by the notification handling process of the PVC in an asynchronous mode.

The notification handling process of the PVC is specified in Figure 7. Upon the reception of a new

[3] Because it refers to the identifier *_eID2*.

monitoring result, the corresponding event and rule are found. If the notification reports a violation, then the handler examines whether the rule has drop actions associated with it and, if so, it forwards the notification to the controller. If this has been the last notification result for the particular rule, then the rule is removed from the drop-action set of the event, so as to release the controller once this drops to the empty set. Finally, the handler performs whatever non-drop actions have been associated with the violated rule.

```
NotificationHandling(NewN: monitorNotif)
  e(m) = NewN.event; R = NewN.rule
  If NewN.Violation Then
    If (R ∈ e(m).D_Rules)
      snd(e(m).chan, NewN)
    If (NewN is last notification for R)
      e(m).D_Rules = e(m).D_Rules - {R}
    apply(NonDropActions(R))
  EndIf
End ResultNotificationHandler
```

Figure 7. Notification handling algorithm

6. Implementation

DVF has been implemented in Java. More specifically, the monitor and the event broker have been implemented in JSE v1.5 using the SIENA event notification service [5]. The PVC has two implementations: one that is based on JSE v1.5 and a version for embedded devices developed on JME-CDC 1.0 and tested on Sony Ericsson's P990i (both simulated and real ones). Both PVCs have been integrated with a peer communication framework that has been developed within the EU project PEPERS and provides basic peer discovery, management and message passing as well as peer authentication management [12].

The DVF can be used by P2P applications that use the PEPERS peer communication framework seamlessly. More specifically, to deploy the capabilities of DVF, developers need to write a monitoring policy, that drives the verification activity during the operation of a P2P application, and provide information about EBr and the monitor(s) that may be used at runtime as part of a DVF configuration file. However, there is no need to add any extra code to their application, unless they want to notify end-users of the monitoring results. In such cases, developers should include code that reacts to the notification messages which are sent by PVCs to peers after rule violations.

7. Related Work

The work presented in this paper is related to two broad strands of research, namely runtime verification and security of P2P systems.

Work in the former strand has the same goal as the framework that we have presented in this paper, i.e. the verification of system properties by monitoring events which are generated during the operation of the system. This strand includes approaches focusing on properties expressed in terms of low level program events focusing mainly on Java programs (e.g. [3][4][6][13][14]) and approaches which focus on systems based on web-services (e.g. [11][17][21]). None of these approaches, however, focuses explicitly on mobile P2P systems and provides a framework that can support effectively the verification of such systems by including mechanisms for: (a) generating events from such systems without having to change their code, (b) negotiating monitoring conditions between peers in order to activate monitoring when a P2P system evolves with the admission and departure of peers, and (c) applying control actions in response to certain types of violations. Thus, the framework presented in this paper is novel in addressing exactly these aspects.

Work in the second area focuses mainly on aspects related to P2P system security rather than dynamic verification, including reputation schemes [7][20], admission control schemes [23][9], techniques for data exchange encryption [24], and decentralised key management [25]. It should be noted that the results of the verification activity performed by the DVF could be used to generate and update peer reputation ratings. Also, through the specification and monitoring of adequate monitoring rules, the DVF could be used to enforce admission control policies.

8. Conclusions

In this paper, we presented a framework that we have developed to enable the dynamic verification of mobile P2P systems. This framework supports:

- the specification of *monitoring policies* to determine *application level properties* that should be monitored in different peers at runtime,
- the *automatic negotiation* between peers at run-time in order to enable the activation of monitoring,
- the *emission of events* required for monitoring from peers to the monitors that perform the checks,
- the *runtime monitoring* of the properties identified in the policies, and
- the *dynamic execution of actions* that need to be taken following the detection of property violations.

The main characteristic of this framework is that it performs dynamic verification based on policies that the owners of individual peers in a P2P system can define. These policies specify properties to be monitored against the operations of not only the specific peer for which the policy is defined but also other peers that may interact with it dynamically at an application level. Policies also specify the actions that should be executed when the monitored properties are violated, and the events that a peer is allowed to emit to other peers interacting with it if the latter want to monitor further properties against its behaviour. This framework has been implemented and tested with hybrid systems of both mobile and non-mobile peers and the performance of the monitors deployed by DVF has been evaluated in elsewhere (see [21]).

Currently, we are working on the transfer of part of a monitor's/EBR's state to other monitors/EBRs, when a peer (or group of peers) moves from one neighbourhood to another. We are also investigating the possibility of extending DVF with a monitor discovery service in which monitors would be treated as a special type of peers that could be discovered dynamically using relevant P2P protocols.

9. Acknowledgements

This work has been supported by the Framework 6 European research project PEPERS (www.pepers.org).

10. References

[1] Androutsellis-Theotokis, S., Spinellis, D., 2004, A survey of peer-to-peer content distribution technologies. ACM Computing Surveys, 36(4):335–371

[2] Barringer, H., et al. 2004, Rule-Based Runtime Verification, In Proc. of 5th Int. Conf. on Verification, Model Checking, and Abstract Interpretation

[3] Brörkens, M. and Möller, M. 2002a, Dynamic event generation for runtime checking using the JDI, In Proc. of the Federated Logic Conference Satellite Workshops, Electronic Notes in Theoretical Computer Science, 70 (4)

[4] Brörkens, M. and Möller, M. 2002b, Jassda trace assertions, runtime checking the dynamic of Java programs, In Proc of Inter. Conf. on Testing of Communicating Systems, Berlin, Germany, 39-48

[5] Carzaniga, A., Rosenblum, D. S., Wolf, A. L., 2000, Achieving scalability and expressiveness in an internet-scale event notification service, In Proc. of the 19th ACM Symposium on Principles of Distributed Computing

[6] Chen, F. and Roşu, G. 2007. Mop: an efficient and generic runtime verification framework. In Proc. of the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications, 569-588

[7] Damiani E. et al., A reputation-based approach for choosing reliable resources in peer-to-peer networks, 2002. Proc. of 9th ACM Conf. on Computer and communications security, 207 – 216

[8] D'Amorim, M., Havelund, K., 2005, Event-based runtime verification of Java programs, In Proc. of 3rd Int. Workshop on Dynamic Analysis (WODA'05)

[9] Fenkam, P. Dustdar, S. Kirda, E. Reif, G. Gall, H., 2002. Towards an access control system for mobile peer-to-peer collaborative environments, WET ICE 2002 – Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises

[10] Freier, A. O., Karlton, P., Kocher, P. C., 1996. The SSL Protocol Version 3.0, Internet draft, available at: http://wp.netscape.com/eng/ssl3/draft302.txt

[11] Ghezzi C., Guinea S., 2007, Runtime Monitoring in Service Oriented Architectures, In *Test and Analysis of Web Services*, (eds) Baresi L. & di Nitto E., Springer, 237-264.

[12] Groce, V. et al. 2007. Framework and Security Verification Tools, Deliverable D6, PEPERS, FP6-26901

[13] Havelund, K., Roşu, G., 2004, An Overview of the Runtime Verification Tool Java PathExplorer, Formal Methods Syst. Des. 24: 189-215

[14] Kim, M., Kannan, S., Lee, I., Sokolsky, O. and Viswanathan, M., 2001, Java-mac: a run-time assurance tool for Java programs, In Electronic Notes in Theoretical Computer Science, 55. Elsevier Science Publishers

[15] Koulouris, T., Tsigritis, T., and Spanoudakis, G., 2006, Dynamic Verification Support Framework, Deliverable D4, PEPERS Project, IST-2004-026901

[16] Mahbub K., 2006. Runtime monitoring of service based systems, PhD, Dep. of Computing, City University

[17] Pistore M, Traverso, P., 2007, Assumption Based Composition and Monitoring of Web Services, In *Test and Analysis of Web Services*, (eds) Baresi L. & di Nitto E., Springer Verlang, 307-335.

[18] Raiciu C., Rosenblum, D. S., 2006, Enabling Confidentiality in Content-Based Publish/Subscribe Infrastructures, In Proc. of IEEE Securecomm '06

[19] Shanahan, M.P., 1999, The Event Calculus Explained, in Artificial Intelligence Today, LNAI 1600:409-430

[20] Song, S., Hwang, K., Zhou, R., and Kwok, Y., 2005. Trusted P2P Transactions with Fuzzy Reputation Aggregation. IEEE Internet Computing, 9(6):24-34.

[21] Mahbub K., Spanoudakis G., 2007. Monitoring WS-Agreements: An Event Calculus Based Approach, In *Test and Analysis of Web Services*, (eds) Baresi L. & di Nitto E., Springer Verlang, 265-306.

[22] Srivatsa, M., Liu, L., 2005, Securing Publish-Subscribe Overlay Services With EventGuard, In Proc. of 12th ACM Conf. on Computer and Communication Security

[23] Saxena N., Tsudik G., Yi J.H., 2003. Admission control in Peer-to-Peer: design and performance evaluation, 1st ACM W. on Sec. of ad hoc and sensor networks, 104-113

[24] Xiaolin, Catania, B. Kian-Lee T., 2003. Securing your data in agent-based P2P systems, In 8th Int. Conf. on Database Systems for Advanced Applications, 55- 62

[25] Law, Y.W., Corin, R., Etalle, S. and Hartel, P., 2003. A Formally Verified Decentralized Key Management Architecture for Wireless Sensor Networks, Personal Wireless Communications, LNCS 2775: 27-39