Methods for Policy Conflict Detection and Resolution in Pervasive Computing Environments

Evi Syukur
SCSSE, Monash University, Australia
evis@csse.monash.edu.au

Seng Wai Loke
SCSSE, Monash University, Australia
swloke@csse.monash.edu.au

Peter Stanski
Telstra Lab, Australia
peter.stanski@stanski.com

ABSTRACT

Recently, there has been increasing work in using policy in pervasive systems. Policy is a relatively new field and much work is still required to explore designs, concepts, and architecture for using policy in pervasive computing environments. In this paper, we briefly introduce the concepts and design of a policy based pervasive system, using Mobile Hanging Services as an example. The main aim of this paper is to investigate several techniques that can be used to statically or dynamically detect and resolve conflicts in pervasive systems. We discuss the conflict detection and resolution techniques in the system as a case study.

Categories and Subject Descriptors

D.2.11[**Software Engineering**]:Software Architectures; H.3.4[**Information Storage and Retrieval**]:Systems and Software; K.6.3[**Management of Computing and Information Systems**]:Software Management.

General Terms

Design, Performance and Management.

Keywords

Policy, Conflict Detection, Conflict Resolution, Web Services, Context, Mobile device, and Pervasive System.

1. Introduction and Motivation

Pervasive computing has a broad view of utilizing computing devices everywhere in the environment and at any time [1]. The idea is that a mobile or non-mobile user can communicate with embedded or non-embedded computing devices, which are invisibly integrated into the environment as soon as s/he steps into that particular space. To date, we have seen a number of pervasive computing systems that have been developed and many of them share similar concepts, although the details of each concept may be different one from another, depending on the target domain of the pervasive system. These basic concepts of the pervasive system are the notions of entities, spaces, services, mobile devices, workstations and contexts.

Recently, there has been increasing work in designing policy based pervasive systems. In our case, policy is used to express a set of rules to govern and control the behaviours of entities in accessing services in specific contexts. Having the additional policy mechanisms in pervasive systems would certainly benefit the user. For example, it allows the users to constrain and control the behaviors of foreign entities operating in his/her environment, and it is used for humans to tell a system what task to do automatically within a certain situation [11]. However, there are some challenges in developing such a system. One of the main challenges we focus on in this paper is detecting and resolving conflicts in an efficient and appropriate manner as they arise in the context of using policies to control mobile services. Conflicts often arise as a result of the differences in policy specifications: e.g., one allows the user to start the service but another prohibits the user from doing so. From our study, we experienced that in pervasive systems, the possibility of conflict occurrence is higher than in other systems (i.e., a distributed system). This is mainly due to a number of contexts and services used, and the mobility of entities, in which, the entity can move freely from one geographical space to another and the entity carries its own rules on how the service should be executed in the designated place.

Due to a number of possible conflicts that may occur in a pervasive environment and each of these conflicts may need different detection and resolution strategies (due to its source of occurrence), we may require a number of techniques to detect and resolve the conflict efficiently. The research presented in this paper attempts to tackle the above issues in our framework for Mobile Hanging Services (MHS). MHS supports policy mechanisms by having and publishing policy software components as Web services. We also propose several techniques for conflict detection and resolution in our pervasive system. We then compare these techniques by considering several aspects of the system such as:

a. System performance - how long it takes to detect or resolve the conflict. The shorter time it takes to detect or resolve the conflict, the faster it is to respond to the user's request (hence, minimizing the user wait time).
b. Implementation - how easy it is to implement such techniques.
c. Accuracy - how often we need to update the conflict detection or resolution result.
d. Does it accommodate all conflicts that may happen in the future?

The rest of this paper is organised as follows. In section 2, we give an overview of the policies in our pervasive system including several possible sources and types of conflicts. In section 3, we describe several general techniques used for conflict detection. In section 4, we discuss general strategies used to resolve the conflict. In section 5, we present a case study: a campus based mobile services system using policy (a MHS application). In section 6, we discuss in detail each of the proposed conflict detection and resolution techniques and compare them. In section 7, we present related work. In section 8, we draw overall conclusions and present future work.

2. Background

This section discusses a definition of policy, followed by an overview of various possible sources of conflicts in pervasive computing environments.

2.1 Definition of Policy

The purpose of the policy is to constrain the behaviours of entities in particular contexts and to ensure that their behaviours (actions performed) are aligned with the rules of the system. A policy language in a pervasive environment can be enriched by

WWW 2005, May 10--14, 2005, Chiba, Japan.

supporting various kinds of normative notions [3,12]. Three basic deontic logic notions that we focus on are:

- Right (R) refers to a permission (positive authorization) that is given to the entity to execute a specified action on the service in the particular context.
- Obligation (O) is a duty that the entity must perform in a given context.
- Prohibition (P) is a negative authorization that does not allow the entity to perform the action as requested in the given context.

2.2 Policy Conflict Sources and Types

In the pervasive MHS system, we may assign different policy specifications to each entity depending on the role that s/he has. Assigning different policy specifications to each user in the system is a way to limit and control the user's behaviours. However, this could also lead to a conflict as the conflict arises due to some differences including:

(a) Policy space modality conflict: conflict occurs as the space (i.e., can be the system space or room space) assigns different specifications on what an entity can do with the service i.e., one allows the user to start the service (i.e., a system) and another prohibits the user from starting the service (i.e., a room) or a room obligates to start a service and at the same time, the user is obligated by the system to stop the service.

These differences lead to a potential or actual conflict that needs to be resolved. In our definition, a potential conflict refers to a conflict that has not happened yet at the time the system detects that such a conflict can happen, as the context or condition for the conflict to occur has not been met. The potential conflict can be further classified into two different types: possible potential conflict and definite potential conflict.

The *possible potential conflict* is a conflict where the possibility of the occurrence is less than the definite potential conflict. This conflict may still not happen even in the right user contexts of location and time. For example, a system allows the user to "start any service" but the room only allows the user to "start media player service". "Any" here means all services which are available for the user in that context. It includes the media player service and some other services in the context. The conflict only occurs if the user starts any service other than the media player service. The conflict will not occur if the user starts the media player service. Hence, we categorize this conflict as a potential conflict with the type *possible*. The *definite potential conflict*, on the other hand, refers to a conflict that will definitely occur if the user is in the right context. For example, a system allows the user to "start media player service" but the room prohibits the user from "starting this service". Once the user is in the right context, this definite conflict will become an actual conflict, as one allows the user and the other prohibits the user.

b) Role conflict: it occurs due to the differences in the privilege that the entity has. For example, one user (with higher privilege) can execute more types of services at any time and any place compared to other users (with lower privilege) who can only execute certain number of services at certain place and time. In our system, the level of privilege is determined based on the level of positions or roles that the user has. As each entity has a different level of privileges, a user with higher level of role may override the execution of the shared service that has been started earlier by a user with lower role. This then leads to a conflict.

c) Entities conflict: it occurs if two or more users have different policy specifications or intentions of what to perform on the service that is running on the same shared resource device. For example, one user wants to start a music service but another user wants to stop this music service which is currently running on the same target machine.

3. Policy Conflict Detection

In this section, we briefly describe goals of conflict detection, followed by several strategies used to detect conflicts in a pervasive computing environment.

3.1 Goals of Conflict Detection

The primary goal of detecting a conflict is to investigate several possible sources of conflicts and types that may occur within the system. Knowing that there is a potential conflict would allow the system to accommodate the conflict resolution earlier. Hence, by the time it occurs, the system is ready with the resolution result. There are also several sub-goals of conflict detection:

a. to group the conflicts based on its type i.e., a possible potential conflict or a definite potential conflict (see section 2.2). This is useful to decide on when to resolve the conflict.
b. to analyse the probability of the conflict occurrence (i.e., normally a possible potential conflict has lower possibility of occurrence compared to a definite potential conflict).
c. to investigate the best technique for conflict detection based on the sources and types of the conflict.
d. to predict the number of occurrences of the conflicts; hence, we can assign the best technique to detect and resolve this particular conflict.
e. to predict the probability of potential conflicts which will become actual conflicts. This is useful to decide when to resolve the conflict. For example, if we can predict that the potential conflict never happens, the conflict resolution for this type of conflict may not be necessary.

3.2 Conflict detection strategy

It is imperative to make a clear distinction on when and where to perform the conflict analysis (conflict detection and resolution), as it can be computationally intensive, time and resources consuming. By analyzing several possible sources of conflicts that may happen in pervasive environments, we propose two different techniques to detect a conflict.

1. Static conflict detection

Static conflict detection aims to detect all types of potential conflicts (possible or definite) which clearly could cause conflicts from the policy specification. This static conflict detection is performed offline on the client side or on the server side. Performing the static conflict detection on the client side is less desirable as it slows down the conflict detection process. This is due to some constraints i.e., limited resources, power and processing speed on the mobile device. The only advantage is the conflict detection result is there on the mobile client side as the user needs it (hence, it does not have to be transferred to the client device). On the other hand, performing static conflict detection on the server side has more advantages compared to the client side i.e., the server (normally a desktop PC) has larger memory size and faster processing speed, and so, can detect the conflict faster. The result can then be pushed onto the mobile client when done.

With static conflict detection, we also need to decide on types of conflicts that we need to detect i.e., whether we only want to detect conflicts which are clearly specified in the policy specification (*predicted potential conflict*) or we want to detect some other conflicts which are not conflicts yet from the policy specification, but they could lead to conflicts if one or more entities are in the space at the right contexts (*unpredicted potential conflicts*). To include all unpredicted potential conflicts will certainly speed up the performance in responding to the user's requests (as it has detected all possible conflicts). The only drawback is it may use up a lot of system resources (i.e., memory and processing speed), as it has to detect the conflict based on all possible combinations of entities, contexts and services that the system has. Moreover, some of the conflict detection results may never be used as the entities may never be in a context as predicted (hence, the conflict may never occur).

Another issue that needs to be taken into consideration is to decide on how often the cached detection result needs to be updated (i.e., if we cache the conflict detection result for future re-use). The detection results may be outdated as perhaps, there are more users registering with the system or some users have modified their policy specifications. To address this issue, several approaches can be incorporated: (a) frequently (i.e., every 5 minutes), (b) periodically (i.e., every Monday) (c) only when the system detects that the user has modified the policy specification or when there is a new user registered with the system.

2. Dynamic conflict detection

Unlike static conflict detection, dynamic conflict detection is performed at run time by dynamically detecting all unpredicted potential conflicts between a number of entities in the given contexts. As dynamic conflict detection is performed some time at run time, the system needs to decide on when to trigger this detection module. We propose five different strategies on when to dynamically detect a conflict.

a. Reactive model

As it is reactive, this dynamic conflict detection is only triggered when there is an explicit request from users i.e., when the user clicks on any action name (start, stop, pause, resume, or submit) from a mobile device to request an action on the service. The detection is done as soon as the system detects that there is a request from the user. If there is a request, the system then collects all the entities' context information and reactively detects the conflicts between those entities in the given context.

This technique is best in the situation with only a few requests from an entity. It takes some time to detect conflicts if there are many requests from the entities. In addition, the detection is only limited to the current location, day, and time, which are related to the requested action and only between the requested user against all other users in the room (not all users in the system).

b. Proactive model

Proactive conflict detection tends to implicitly and automatically detect the conflict by sensing the user's current context i.e., when the user moves in or out of the room. This technique is best used in the situation where performance is paramount. The proactive conflict detection detects all the potential conflicts that may occur in the given context and may cache the result for future re-use. The proactive technique is also considered as pessimistic conflict detection. We are pessimistic that there will be a conflict between those entities in the room, as each entity may try to perform different actions for the same shared service. Hence, the system proactively catches all potential conflicts that may occur in the given context. In addition, this technique is considered useful only if the participating entities (i.e., users) are still in the same context where the conflict is predicted to happen. If one of these entities has moved to a different location, the predicted potential conflict may no longer be an actual conflict (as this type of conflict only occurs if two or more entities which have different specifications on the same target service are still in the same space).

Moreover, there are two issues that need to be addressed in order to increase the accuracy of dynamic conflict detection result:

- What happens if in the middle of process of detecting a conflict, another user comes in? Will the system continue with the detection process? If it continues, it then has to re-compute the result after some time, as it is already outdated.
- What happens if the user has left the space and this user is already in the conflict detection list result? Do we need to remove him/her from the list? What happens if s/he comes back to the space after some time? We need to know when to remove users from the list. Also, there is a problem, if we keep all the results in the memory, as the server may be overloaded with outdated results and perhaps, there is no longer a conflict between users (as one of the conflicted users is no longer in the space).

c. A combination of reactive and proactive models

A combination of these techniques is useful when we want the system to act proactively in a certain situation i.e., in an examination room, a seminar room and in a certain place, it acts reactively i.e., in the individual room. This is mainly because, at a public place, there are many users coming in and out of the place, therefore, it is useful to employ a proactive conflict detection technique here. At the individual room, usually, only the owner with some other visitors that may not perform many activities, hence, we detect the conflict reactively. A decision to choose whether to perform a proactive or reactive behavior can be based on: **(i) the location** i.e., proactive in public place and reactive in the individual place (i.e., a user's office). **(ii) the day and time** i.e., on Monday at any place, proactively detects the conflict, because, it may be a busy day and many students come to the University or at the shopping centre, there may be a lot of visitors visiting the mall, but, other days, we detect the conflict reactively. **(iii) the number of users in the location**. For example, if the system detects there are more than five users in the location, a proactive behaviour is used. However, if there are less than five users, the system then detects the conflict reactively.

d. Predictive model

Predictive model detects the conflict based on the user's history file. By analyzing the user's history file, the system can predict the user's movement and the person that the user is going to meet. For example, from the history file, user A is always going to room B and meeting user B on Wednesday at 12PM. Based on this information, the system may want to compute the conflict detection proactively between these users (user A and B) at room B. This technique is considered useful only if the system prediction is correct (i.e., the user always does the same activity as listed in the history file). However, if the user's movement and activity are not anticipated by the system (i.e., the user is moving to a different room and meeting different people), there will be a delay in responding to the user's request. This is due to the conflict detection result which has been previously computed is irrelevant to the user's current context. Hence, the system will need to re-detect the conflict based on the user's current location, day, time and people that s/he is meeting.

4. Policy Conflict Resolution

When there is a potential or actual conflict detected by a conflict detection module, it becomes necessary to resolve the conflict. Several aspects discussed in this section are the goals of the conflict resolution, when and how to resolve conflicts, as well as when to update the conflict resolution result.

4.1 Goals of Conflict Resolution

The primary purpose of conflict resolution is to resolve all types of conflicts in minimum amount of time, and so, minimizes the user wait time. Several sub-goals of conflict resolution are:

a. to investigate several techniques on how to resolve the conflict based on its sources and types.
b. to decide when it is the best time to resolve the potential or actual conflicts.
c. to monitor whether the conflict resolution result satisfies both of the conflicted entities. If the conflict resolution result does not satisfy the conflicted entities, we need to think of the best solution that will benefit both of these entities i.e., allowing the conflicted entities to challenge the system and resolving the conflict by taking into account the user's current situations.
d. to decide on how often the conflict resolution module needs to be re-computed.
e. to analyse whether the conflict resolution result is useful (i.e., the conflict resolution result will be used at run time, as the predicted potential conflict becomes an actual conflict).

4.2 Techniques to resolve the conflict

We propose several conflict resolution techniques to handle possible conflicts that may occur in pervasive systems. Some additional resolution techniques or further refinements of each of the following resolution techniques are required depending on the target pervasive domain. This paper discusses only the major conflict resolution techniques which can be used across pervasive systems that employ and share the basic pervasive concepts as discussed earlier in introduction. These resolution techniques are **(a) Role hierarchy overrides policy.** The role hierarchy overrides policy is used if the conflict occurs between users who have different roles, in which a user with a higher role can override the policy that belongs to the user with a lower level of role. **(b) Space holds precedence over visitor.** This technique is used if a conflict occurs between a user and a room. For example, the system permits a user to start a service at room A, but room A prohibits the user from starting this service. If there is a conflict, the room (representing its owner) always wins, regardless of the levels of role of the visitor. **(c) Obligation holds precedence over rights.** This technique is used if a conflict occurs between an obligation and the right. An obligation always wins over the right. For example, a user is permitted by the system to start a media player service, but a room obligates the user to stop this service.

4.3 When to resolve the conflict

We propose two strategies on when to resolve the conflict in pervasive computing environments.

a. At the time when a conflict is detected

This is a *pessimistic conflict resolution* technique. We are pessimistic that some or all detected potential conflicts will become actual conflicts. Hence, the system resolves all conflicts immediately as soon as the system detects them. Depending on the conflict detection technique that the system employs, with this technique, the conflict can be resolved offline (i.e., when users are not in the space yet) or at run time. For example, if we employ a static conflict detection technique, the conflict resolution of all potential conflicts is done offline, as soon as they are detected. However, if the system employs a dynamic conflict detection (i.e., a reactive technique), the conflict resolution is only performed at run time, as the conflict is only detected at run time.

In addition, with this technique, we can further choose which conflicts to resolve based on its type such as: **(i) Resolve only a definite potential conflict:** The technique here resolves only a definite potential conflict, as we are sure that it will become an actual conflict once the entities are in the right contexts for the conflict to happen and resolve the possible potential conflict only when the contexts for the conflict to happen are met. This technique does not anticipate all resolution results. Hence, it may experience a delay in responding to the user's request, especially if the possible potential conflict happens to be an actual conflict at run time. **(ii) Resolve both possible and definite potential conflicts**. The system can also choose to resolve both types of conflicts as soon as they are detected. These potential conflicts are solved, though they have not happened yet to be actual conflicts. This technique would minimize the user wait time, as it has resolved all predicted conflicts prior to become actual conflicts. However, if none of the predicted conflicts become actual conflicts, it may waste the system resources.

b. At the time when the potential conflict becomes an actual conflict (normally at run time)

This is an *optimistic conflict resolution* technique. We resolve the potential conflicts just when they become actual conflicts. We do not resolve these potential conflicts, just when we detect them, as we are optimistic that the conflicts that we have detected may or may not become actual conflicts. This is due to several factors such as the user may not be in the context where the conflict is detected to happen or the user does not execute the service in the specific context (i.e., specific location, day and time) where a conflict can arise (although, it is clearly a conflict from the policy specification). For example, the user is allowed by the system to start a media player service at any day, however, the room only allows the user to start this service on Monday only. We are optimistic that the conflict here will not happen, unless the user starts the service on any other days (other than Monday).

4.4 How often to update the conflict resolution result

It also would be good to cache the conflict resolution result for future re-use. The question here again, we need to decide on how often the cached result needs to be updated. One simple solution is to update each time the conflict detection module is re-computed (when the cached conflict detection result is updated).

5. Case Study

This section discusses in detail on how policy specification, conflict detection and resolution strategies are used in pervasive computing environments. One sample prototype that we have developed is a campus based policy system within MHS.

5.1 MHS on Campus

As discussed earlier in introduction, our definition of a pervasive computing environment consists of entities, spaces, services, mobile devices, workstations and contexts. The details of each of these concepts depend on the target pervasive system and its environment. For example, an entity in a campus domain refers to a student, a lecturer and a head of school, however, in a shopping mall domain, it could mean something different i.e., a customer and a seller. In this section, we mainly focus on the pervasive concepts and policy specifications in a campus domain. We describe each of these concepts as follows:

a. **Entities.** Entities here refer to mobile users which are always on the move (move from one geographical space to another). Three types of entities in our system are a student, a lecturer, and a head of school. By default, our **s**ystem imposes certain **r**ights (denoted by sRe), **o**bligations (sOe) and **p**rohibitions (sPe) to each of these **e**ntities depending on the role that the entity has and the physical space that the entity is visiting. In addition, each of the **e**ntities in the system can also impose a certain **o**bligation to the

system (eOs), created via a user policy application that we have. In summary, each of the entities in the system will have:

sRe_i, sOe_i, sPe_i and e_iOs

Note: i denotes a specific user i.e., user **i**.

b. **Spaces.** Spaces here can be a physical room that is represented by a geographical location e.g., room B558. The room entity has its own policy that can be used to restrict the visitors' behaviors or actions on mobile services in the room. Generally, the room's policy is created by the owner of the room. The public place in our system (e.g., tea room, corridor, or seminar room) is owned by the system. Hence, the public policy is created by the system (i.e., a developer/system administrator).

c. **Services.** A service refers to a software tool that is enlisted as users need it and it helps users to accomplish the tasks by downloading the service application or mobile code onto a target machine (i.e., a mobile device or a desktop PC machine). We have two types of services in our system: a shared resource service e.g., Mobile VNC [10] and Mobile Media Player applications [11], in which the service is downloaded onto a shared desktop machine and it can be controlled and accessed by all legitimate users from their mobile devices in that specific location. A non-shared resource service, on the other hand, is a service that is downloaded to and compiled in the user's mobile device (e.g., a Mobile Pocket Pad Service [9] and is only accessible by that user).

d. **Mobile Devices** i.e., handheld devices which display service interface and can execute service processes.

e. **Workstations**. It can be a normal desktop PC where services can be executed (run) or a server that hosts all context-aware and policy related components.

f. **Contexts**. Contexts are conditions that must be met before a list of services can be displayed on the mobile device or before the user's request to perform an action is approved. In our work, contexts consist of a user's identity, location, day and time.

5.1.1 Architectural Design

Our policy software components handle the user's request to perform some actions on the service. The request can be start, stop, pause and resume the service. This section provides a high level architecture and description of these parts of our mobile policy based framework (see Figure 1 below).

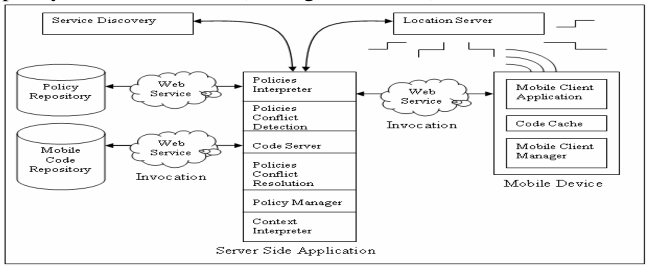


Figure 1: High Level Architecture of MHS Policy Framework

The details of each of our context-aware software components have been discussed in [9]. We now describe each of our policy software components: **(a) Mobile client query manager (on the mobile client side).** It handles the request from the user and sends this request to the policy manager. **(b) Policy manager (on the server side).** Policy manager manages the interaction between the mobile client and the server, in which the mobile client sends a request to the policy manager and the policy manager computes the request and returns the result back to the client. The result is either allowing or disallowing the mobile user to perform the action. **(c) Policy interpreter (on the server side).** The policy interpreter component specifies a set of rights, prohibitions and obligations which are useful for the user in the particular contexts. **(d) Policy conflict detection module.** The policy conflict detection detects lists of potential or actual conflicts that may occur between entities in the system. **(e) Policy conflict resolution module.** The policy conflict resolution module handles conflicts between entities in the system.

5.1.2 Prototype Implementation Details

We present our prototype implementation where we have implemented some of the conflict detection and resolution techniques discussed in previous sections. Our MHS system consists of users with mobile devices who are always on the move, a web service that determines the user's current location, and policy software components which handle a user's request to perform an action on a particular service.

As for conflict detection, our system employs a combination of static and dynamic conflict detections. Static conflict detection is performed offline on the server side and statically checks the entity's policy specification to detect the policy space modality conflicts (i.e., between a policy specification from a system to the user and from a room to the user). The policy space modality conflict may occur here, as the system may permit the user to "start the service", but the room prohibits the user.

We also cache this static conflict detection result for future re-use. When the system detects there is a new user added or there is a user modified his/her policy, our static conflict detection module then updates the cached result. Our dynamic conflict detection further detects the conflict at run time (i.e., a conflict between users). We only detect conflicts between users at run time, as in pervasive systems, the user is always on the move and the movement is unpredictable; hence, we do not know where the user is going to and whom s/he is meeting. Therefore, it would be good to detect this type of conflict dynamically at run time (just when the users are already in the space).

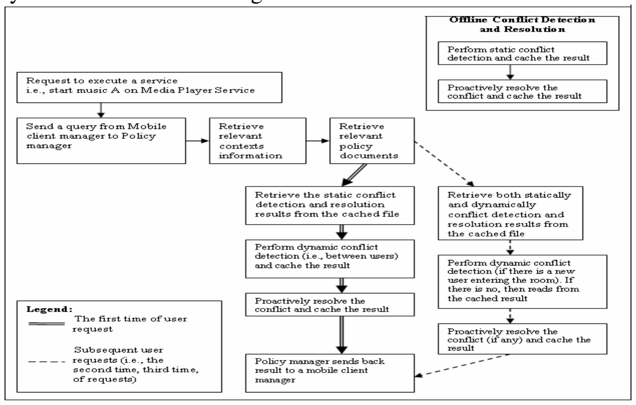
Before further checking for conflicts between users, dynamic conflict detection first detects the type of the service that a user would like to perform. If it is a shared resource service, then the dynamic conflict detection needs to check whether there is a conflict between one user's policy against another user's policy. Checking between users' policies are required for shared services only, as the service is running on the shared machine that allows any legitimate user to control the execution of the service from his/her mobile device. If it is a non-shared resource service, the dynamic conflict detection does not need to further check the conflict between users as the non-shared resource service does not involve other users (only between a user and the room). As we have already detected the conflict between a user and a room statically, the dynamic conflict detection for the non-shared service then just reads from the cached detection file.

Once the checking on the type of the service is done, the dynamic conflict detection then needs to read and process the cached result to find out whether the user is permitted by the system to perform the specified action. If so, then it checks whether the user is permitted to perform the action by the room. If the user is permitted, the system then continues to perform dynamic checking whether there is a conflict between users if the specified action is performed. We use a combination of static and dynamic conflict detections in order to speed up the conflict analysis and processing time. Hence, it will reduce the user wait time. Employing only a single conflict detection strategy i.e., only a static or dynamic conflict detection would slow down the system performance. In addition, our system also resolves all potential conflicts as soon as they are detected. Resolving the conflict only when it becomes an actual conflict will result in delay in responding to the user's request.

It would be preferably to detect and resolve the conflict statically (offline). However, due to undiscovered all potential conflicts at this time, as some of the conflicts may only occur if a number of entities are in the contexts, run time conflict detection and resolution are also necessary. However, there is still a challenge here in deciding on what types of conflicts should be handled statically or dynamically when considering the aspects of system resources and performance. For example, detecting and resolving all conflicts statically can certainly improve the system performance (as the system has anticipated all potential conflicts with their resolution results). However, detecting and resolving all conflicts statically also has a drawback, in which, it may use up a lot of system resources and may waste the resources, especially if the predicted conflicts never become actual conflicts (the detection and resolution results are never used). This area is still an ongoing work that needs to be further explored in the future.

5.1.3 Performance Results

The framework has given promising results in obtaining a list of policies which are useful to the user, detecting and resolving the conflict both offline and at run time. The evaluation starts from the Web service call to get a user's policy up to resolving the conflict and deciding whether the user is permitted to perform the action on the specified service. The evaluation aspects of our system are described in Figure 2 below.



Offline Conflict Detection and Resolution
Perform static conflict detection and cache the result
Proactively resolve the conflict and cache the result
Request to execute a service i.e., start music A on Media Player Service
Send a query from Mobile client manager to Policy manager
Retrieve relevant contexts information
Retrieve relevant policy documents
Retrieve the static conflict detection and resolution results from the cached file
Retrieve both statically and dynamically conflict detection and resolution results from the cached file
Perform dynamic conflict detection (i.e., between users) and cache the result
Perform dynamic conflict detection (if there is a new user entering the room). If there is no, then reads from the cached result
Proactively resolve the conflict and cache the result
Proactively resolve the conflict (if any) and cache the result
Policy manager sends back result to a mobile client manager
Legend:
The first time of user request
Subsequent user requests (i.e., the second time, third time, of requests)

Figure 2: Evaluation aspects

In our evaluation and testing, results were collected for five times of requesting the system to execute the same action with the same service name at the same contexts i.e., a mobile user requests to start a media player service with a particular song name on Saturday, between 12-2PM at B558 room. We measure each of the evaluation aspects above for five times of policy execution, assuming the number of policies in the location are the same throughout the execution i.e., two policies exist in the location – a user's policy and a room's policy. There is also one conflict found between a user and a room, in which a room does not allow a user to perform such a service on Saturday, between 12-2PM at B558 room.

The evaluation results are illustrated in Figure 3 below. These figures were obtained on an iPAQ emulator that is running on the laptop using wireless Wifi network for internet connection. From Figure 3, we can see that the time required to call the Web service: send a query from a client to policy manager, retrieve context information, retrieve relevant and parse policy document, read from the cached results and send back result to the mobile client manager decreases for the 2nd, 3rd, 4th, and 5th times of web service calling. The first call of the Web service takes longer time, as the system needs to compile and download the local host Web service proxy object to the device.

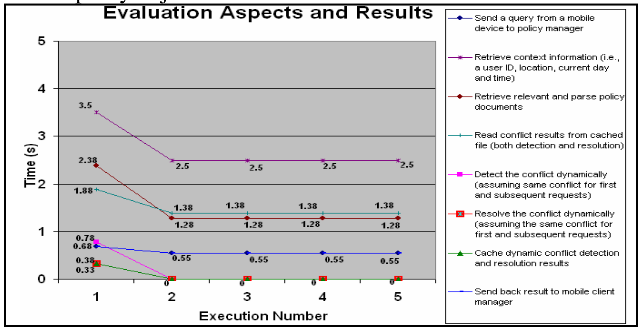


Figure 3: Experimental results

The proxy object allows the Web service to be treated like other .NET classes. The 2nd and subsequent calls to the web service will have much shorter times as it reuses the service proxy object already on the local mobile device. The amount of time required to perform static conflict detection and resolution at compile time is 3.18s (=1.17+0.48+1.05+0.48). Here, the static conflict detection component first detects whether the system gives a user permission to execute the service. If the system permits the user, we then continue checking with the room's policy (i.e., whether the room permits the user to execute the service). Here is the formula to detect and resolve the conflict statically.

$T_{\text{static conflict analysis(s)}} =$

$T_{\text{detect a conflict statically}} + T_{\text{cache the conflict detection result}} + T_{\text{proactively resolve the conflict}} + T_{\text{cache the conflict resolution result}}$

If there is a permission given by the room, we continue checking it against other users' policies or a room's obligation (conflict detection at run time). Here, it takes 0.78s to detect a conflict at run time. We found one conflict between a user and a room's obligation. The dynamic conflict detection module here only checks the conflict against a room's obligation, as we only have one user in the location. Checking against a room's obligation is necessary because the room also imposes a certain duty to the user. Hence, we want to ensure that there is no conflict between the user's action and the room's obligation.

This static and dynamic conflict detection results are also cached on the server for future re-use. The second and subsequent policy execution of the same action, service and contexts will just read from the cached file (assuming there is no user moving in or out of a place). Therefore, the dynamic conflict detection time for subsequent policy executions here is zero. Having static conflict detection would help to minimize the user wait time by detecting all potential conflicts between a user and room offline. Detecting such conflict at run time would consume lots of time. Hence, it is recommended to detect it statically, although some of the conflict detection results may not be useful as some of the users may not be in the context as predicted.

As we employ a proactive conflict resolution strategy (resolving conflicts as soon as the system detects them) for both static and dynamic conflict detection, the system takes shorter time to resolve some other detected dynamic conflicts at run time. It takes 0.33s to resolve the dynamic conflict for the first time a service is called. Our system also caches the dynamic conflict resolution result on the mobile device. Hence, the second and subsequent requests of the conflict resolution for the same conflict that has the same action name, target service and contexts would just read from the cached file. In addition, the time it takes to cache the results (i.e., conflict detection and resolution results) at run time is 0.38s (for the first time requesting the service). As there is no conflict occurring for subsequent requests, there is no result that needs to be cached (0s time to cache results for subsequent requests). Finally, we present a formula to calculate the time required to request to perform an action on a shared or non-shared resource service till the system responds back to the user. This formula is illustrated as follows:

$$T_{\text{user wait time(s)}} = T_{\text{send a query from a mobile client to a policy manager}} + T_{\text{retrieve context information}} + T_{\text{retrieve and parse relevant policy documents}} + T_{\text{read conflict results from a cached file (both detection and resolution)}} + T_{\text{detect a conflict dynamically (if any)}} + T_{\text{resolve a conflict dynamically (if any)}} + T_{\text{cache results (if any)}} + T_{\text{send back result to the mobile client manager}}$$

Based on the formula above, we can conclude that the worst-case scenario for the user wait time is the first time of requesting the service, which takes 10.61s (= 0.68 + 3.5 + 2.38 + 1.88+ 0.78 + 0.33 + 0.38 + 0.68). The 3.5s is the total time to retrieve context information. It takes 3s to get a user's current location using Ekahau location tracking system via a Web service call and 0.5s to retrieve a user identity, current day, and time. The 3s Ekahau delay can be eliminated, if we assume the user is still in the same location (for the first and subsequent requests), and so, the system does not need to re-detect the user's current location.

The best case scenario i.e., the minimum time delay to get a response back from the policy manager is in any execution which is not the first. In such a case, the delay time is 6.26s (=0.55 +2.5 + 1.28 +1.38 + 0 + 0 + 0 + 0.55) – assuming the location context for subsequent requests are still the same. The delay time to detect subsequent requests decrease to 6.26s, because, the Web service calls in subsequent requests, re-use the local proxy object, which has been downloaded and compiled previously and also the subsequent requests do not require to perform dynamic conflict detection and resolution (only read from the cached file) as the conflict is the same as in the first run.

6. Discussion

We observe that each of the proposed conflict detection and resolution techniques has its own advantages and disadvantages, such as: 1) Static conflict detection: it accommodates all potential conflicts that may happen in the future (hence, it will speed up the performance in responding to the user's requests), is simple to develop and relatively easy to maintain. However, this technique only suits if the number of entities in the system is not too many and policy specification and number of entities in the system are relatively static. More entities mean more policy specifications which mean more policies to compare. Allowing entities to modify his/her policy specification at run time or having a new user registered, requires the system to update the static conflict detection result which has been previously computed. Hence, it will use up a lot of resources and may be quite tedious, as it has to re-detect the conflict between all entities in the system. Moreover, some of the conflict detection results may never be used as the entities may never be in the context as they are predicted - hence, the predicted potential conflict never becomes an actual conflict.

2) Reactive based Dynamic Conflict Detection: this technique takes shorter time to detect all potential conflicts in the given context as it only checks the conflict between the requester and number of users in the room. It is also simple to develop and maintain and suits any situation (i.e., static/dynamic policy specifications or entities) as the conflict detection is triggered reactively i.e., when there is a request from a user to perform an action on the service. The main drawbacks of this technique are long delays in detecting and resolving the conflict between entities, as the system only starts to detect and resolve the conflict when there is a request from the user. Moreover, detecting the conflict based on the user's request may not be a good idea as one user may request (click on the action name) more than once in a minute i.e., user A clicks on the start button twice and user B clicks on the stop button three times, hence, the system needs to execute the conflict detection for five times.

3) Proactive based Dynamic Conflict Detection: this technique accommodates all potential conflicts in the given context (hence, reduces the user wait time), use less system resources (memory and CPU processing) compared to the static conflict detection technique, as it only detects conflicts between entities which are in the same context (not all entities in the system). It is also considered easy to develop and suits for any situation with static or dynamic policy specifications or entities. However, the system maintenance can be challenging, as we need to know the best time to update the conflict detection result (when to proactively detect a conflict) i.e., when the system detects that there is a new user moves in or out of the space, frequently every 5 seconds, or when the system detects there are more than certain number of users in the space such as more than two users in the room.

4) A combination of Reactive and Proactive based Dynamic Conflict Detection: this is an ideal technique among all other conflict detection techniques. It accommodates all potential conflicts in the given space by using a combination of reactive and proactive techniques. It can be proactive in some situations and reactive in others, and so, can further reduce the system resources (memory and CPU processing). It also suits in any situation (with static or dynamic policy specification, entities, services and contexts). This technique is also easy to implement. The only issue here is we need to decide when and under which situation a proactive or reactive behaviour should be performed.

5) Predictive based Dynamic Conflict Detection: this technique is much more complex to develop and maintain and does not accommodate the user's unpredictability.

In addition, we found that the potential conflicts which are detected at run time by using a reactive technique have higher possibility to become actual conflicts compared to other techniques (i.e., a proactive or predictive technique). This is mainly because in reactive technique, the detection is only performed when there is a request from a user and the detection is looking for conflicts only for the current day, time and location (hence, if there is a conflict found, the contexts for the conflict to occur must have been met). On the other hand, a proactive technique proactively detects all potential conflicts between users although the contexts for the conflict to occur have not been met.

Moreover, for conflict resolution, the best technique is to have a proactive conflict resolution strategy that immediately resolves the conflicts as soon as the system detects them. This technique anticipates all potential conflicts that may happen between entities in the future. Hence, it improves the system performance and certainly minimizes the user wait time. However, some of the conflict resolution results may not be useful as some of the detected potential conflicts may never happen at run time.

7. Related Work

This section provides a brief overview about the research work that has been done to date that also concentrates on exploring different strategies used to detect and resolve conflicts in policy systems. Some earlier policy work in pervasive systems are Rei [3], Spatial Policies [4] and Policy for Agent Mobility work [8]. In addition, only few work done to date explores different strategies of policy conflict detection and resolution. A notable project is a work done in [5,6,7] that explores different techniques used for conflict detection and resolution in enterprise and management policy based systems. Our conflict detection and resolution techniques to some extent have similar philosophy to this project. The only difference is the target environment, we focus on pervasive systems which have services, entities, contexts, mobile devices, workstations and spaces.

As our system is designed for pervasive computing environments, in which users are always on the move and often require immediate response from the system of their requests, the sources and types of conflicts found in our system are also different from the one in [5,6,7]. This then leads to some differences in designing and implementing the conflict detection and resolution techniques. For example, we have conflicts on permissions, obligations and prohibitions between mobile users, as well as between a mobile user and the space. In contrast to [5,6,7], they do not take into account the mobility of users and the notions of services, and so, the conflicts found in the system are mostly between non-mobile users who are trying to access system or a user's resources information. In addition, our pervasive system tends to focus more on the system performance that aims to deliver the service, detecting and resolving conflicts in minimum amount of time.

8. Conclusions and Future Work

This paper has presented a design, model and architecture of a policy based framework in pervasive environments. We have proposed several techniques or strategies for conflict detection and resolution. We also have implemented and tested our policy system with some of conflict detection and resolution strategies on the mobile emulator that runs on an 802.11b wireless network. While implementing some of the conflict detection and resolution strategies, we discovered that each of the proposed strategies both for conflict detection and resolution offers some advantages and disadvantages. The suitability of each strategy is dependent on the system situations (i.e., number of entities, physical rooms, contexts, types of services and target services that the system employs), the system goals (i.e., it aims for high performance, so requires a comprehensive and more complex conflict detection (i.e., a predictive model) and resolution modules), and types of conflicts that the system attempts to detect or resolve (i.e., we tend to detect all policy space modality conflicts statically).

Moreover, we have experienced that using a combination of static and dynamic conflict detection helps to improve the system performance (minimize users wait time), rather than only using a single detection technique (i.e., static only or dynamic only). We also found that resolving all potential conflicts (possible or definite conflicts), as soon as they are detected, would certainly reduce the delay in responding to the user's request, and so improve system performance.

A number of aspects of future work that need to be further analysed, explored and developed are: a) Continue working on proactive and predictive conflict detection strategies. b) Allowing users to modify their policy specifications dynamically at run time. c) Apply our policy concepts (i.e., designs, conflict and detection and resolution strategies) in different pervasive environments or domains i.e., a museum gallery, shopping mall, airport. d) Monitor the probability of potential conflict occurrence e) study the nature and complexity of each conflict found in pervasive systems, also finding out how much of memory, CPU cycles required to detect and resolve conflicts both statically and at run time.

9. References

[1] Weiser, M., "The Computer for the 21st Century", *Scientific American*, 9 1991.

[2] Chen, G. and Kotz, D. (2000), "A Survey of Context-Aware Mobile Computing Research", Dartmouth Computer Science, *Technical Report TR2000-381*.

[3] Kagal, L., Finin, T. and Joshi, A., "A Policy Language for a Pervasive Computing Environment, *Proc. of IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, Italy, June 2003.

[4] Scott, D., Beresford, A. and Mycroft, A., "Spatial Policies for Sentient Mobile Applications", *Proc. of IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, Italy, June 2003.

[5] Dunlop, N., Indulska, J. and Raymond, K., "Dynamic Policy Model for Large Evolving Enterprises", *Proc. 5th IEEE Enterprise Distributed Object Computing Conference*, Seattle, Sept 2001.

[6] Dunlop, N., Indulska, J. and Raymond, K., "Dynamic Conflict Detection in Policy-Based Management Systems", *Proc. 6th IEEE Enterprise Distributed Object Computing Conference*, Lausanne, Sept 2002

[7] Dunlop, N., Indulska, J. and Raymond, K., "Methods for Conflict Resolution in Policy-Based Management Systems", *Proc. 7th IEEE International Enterprise Distributed Object Computing Conference*, Brisbane, Sept 2003, pp 98-109.

[8] Montanari, R., Lupu, E. and Stefanelli, C., "Policy-based dynamic reconfiguration of mobile-code applications", *IEEE Magazine*, July 2004.

[9] Syukur, E., Cooney, D., Loke, S.W. and Stanski, P., "Hanging Services: An Investigation of Context-Sensitivity and Mobile Code for Localised Services", *Proc. of the IEEE International Conference on Mobile Data Management*, USA, Jan 2004, pp.62-73.

[10] Syukur, E., Loke, S.W. and Stanski, P., "The Mobile Hanging Services Framework for Context-Aware Applications: the Case of Context Aware VNC", *Proc. WIS Workshop*, Portugal, April 2004.

[11] Syukur, E., Loke, S.W. and Stanski, P., "A Policy based framework for Context Aware Ubiquitous Services", Proc. of the Embedded Ubiquitous Computing Conference, Japan, August 2004, LNCS, vol. 3207, Springer-Verlag, pp.346-355, 2004.

[12] Mally, E. "The Basic Laws of Ought: Elements of the Logic of Willing", 1926.