Handling Policy Conflicts in Call Control

Lynne BLAIR [1], Kenneth J. TURNER
Computing Science and Mathematics, University of Stirling, Stirling, FK9 4LA
lb@comp.lancs.ac.uk, kjt@cs.stir.ac.uk

Abstract. Policies are becoming increasingly important in modern computer systems as a mechanism for end users and organisations to exhibit a level of control over software. Policies have long been established as an effective mechanism for enabling appropriate access control over resources, and for enforcing security considerations. However they are now becoming valued as a more general management mechanism for large-scale heterogeneous systems, including those exhibiting adaptive or autonomic behaviour.

In the telecommunications domain, features have been widely used to provide users with (limited) control over calls. However, features have the disadvantage that they are low-level and implementation-oriented in nature. Furthermore, apart from limited parameterisation of some features, they tend to be very inflexible. Policies, in contrast, have the potential to be much higher-level, goal-oriented, and very flexible.

This paper presents an architecture and its realisation for distributed and hierarchical policies within the telecommunications domain. The work deals with the important issue of policy conflict – the analogy of feature interaction.

1. Introduction

Features have long been used as units of client-valued functionality within the telecommunications industry. One key benefit is their ability to facilitate evolutionary development, for example accommodating new requirements or functionality in existing systems by incorporating new features. Feature-driven development is also popular in software product-lines, where a product is structured in a way that allows common units of development (e.g. features) to be shared, thus enhancing flexibility and software re-use, and ultimately reducing development costs [Nyholm02].

Inherent in such feature oriented systems is the problem of feature interactions, where the presence of one feature in a system affects the behaviour of another. Whilst this behaviour may be a desirable effect for some features, there are other instances where the interaction causes undesirable behaviour. Many examples of this have been identified within telecommunications systems, along with numerous techniques for the detection and resolution of such interactions. These are well documented in the series of “Feature Interactions in Telecommunications and Software Systems” proceedings, e.g. [Calder00] and [Amyot03]. Within this domain, surveys of feature interaction analysis techniques can be found in [Keck98] and [Calder03].

However, feature interaction is only one small part of a much more general problem of interactions, or conflicts, throughout the overall software development life-

[1] Carried out while on sabbatical from Computing Dept, Lancaster University, Lancaster LA1 4WA

cycle. The study of requirements interactions [Robinson03], rule or goal conflicts [vanLamsweerde98] and policy conflicts [Lupu99][Dini04] are all areas of research in their own right. In this paper we focus on the use of policies for call control, and on issues surrounding the resolution of conflicts between such policies. Of necessity, simple examples are used to illustrate the approach. However as can be seen from related publications, the policy system is capable of great expressiveness and subtlety – well beyond what traditional features can achieve.

2. The Use of Policies for Call Control

Policies, within the computing domain, are generally regarded as being a high-level, user-oriented and flexible mechanism for controlling a system or a set of system parameters. Policies have proven beneficial in many different areas, including managing access control or security concerns, controlling collaborative systems, managing acceptable limits for quality of service parameters, or more generally managing networks.

Recently, researchers have also started to investigate the use of policies (or scripts) to facilitate call control, e.g. [Bertino02], [Chentouf03] and [Nakamura03]. As discussed above, control in the communications domain has traditionally been achieved by a more technology-driven approach, e.g. through low-level and fairly inflexible units of functionality known as features. More recently, service-oriented approaches have placed the emphasis on the service provided to the user rather than on the features needed to support the service. Policies do not replace these, but rather can be viewed as an overlay on them, providing users with a mechanism to configure and manage their own call control environment. The ACCENT project (Advanced Call Control Enhancing Network Technologies) has focused on the use of policies in this way.

In the following section we provide background information on the architecture developed to support the use of policies in call control, and then return to examples of policies in sections 3 and 4.

Policy System Architecture

Although this paper discusses policy support related to call control, the architecture is generic and has been designed for ready extension to other application domains. Any reasonable communications layer can be used. So far, policy support has been developed for various forms of Internet telephony: SIP (Session Initiation Protocol) [Rosenberg02], H.323 [ITU00] and PBX (Private Branch Exchange) [Mitel04]. Extensions to other forms of telephony are possible, e.g. the IN (Intelligent Network) and mobile telephone networks. Investigations into policy support for H.323 (carried out in parallel with the ACCENT project) are reported in [Huang05].

The policy system is supported by the (extensible) policy language APPEL (ACCENT Project Policy Environment/Language) [Reiff-Marganiec04c]. The core of APPEL is common to many applications. APPEL is then extended with application-specific vocabularies. In the work described here, the extension is specific to call control.

The policy system architecture is shown in figure 1. In this figure, double-headed arrows indicate that many instances of a system may appear at this end. For example, one policy server may support many communications servers. In turn, one policy store

may support many policy servers. All the arrows represent socket connections, so the system can be truly distributed. The systems are logically separate, but may be one physical system. Further details of the policy system architecture can be found in [Reiff-Marganiec04b], and other aspects of the ACCENT project in [Reiff-Marganiec04a], [Reiff-Marganiec04d] and [Turner04].

The novelty of this paper over these previous publications lies in the use of APPEL not only for basic call control, but also for the resolution of policy conflicts. As will be explained, resolution policies are processed in an almost identical way to call control policies, and importantly are handled within the same policy system architecture.

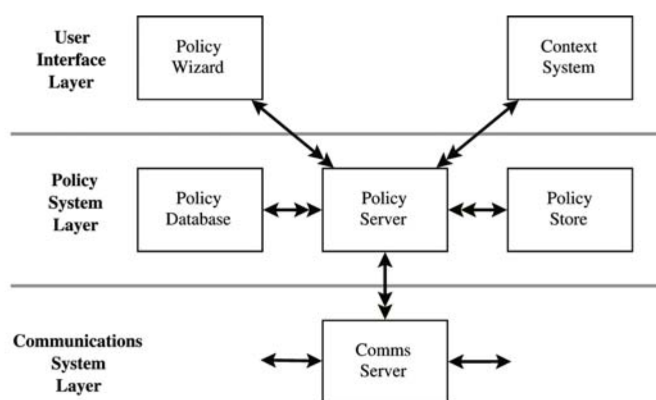

Figure 1. Policy System Architecture

The architecture maintains a clear separation between three layers: the communications systems layer containing the raw networks, the policy system layer dealing with policy storage and execution, and the user interface layer providing direct end user support.

Communications System Layer. Minimal assumptions have to be made about the networks to be enhanced with policies. It is presumed that each network will have communications servers where calls are processed. Communications servers take various forms such as a proxy server for SIP, a gatekeeper for H.323, or an SSP/SCP for the IN. A policy module needs to be written for each communications server using its API; so far, four communications server modules have been developed for policy support. Such a module maps between network-specific messages and neutral policy messages. Environment information is extracted from call requests as they are processed (e.g. caller, callee, call subject). This information is sent in a generic format to the associated policy server, which responds with generic actions, taken from the policies, as to how the call should be handled (e.g. continue as normal, add a party to the call, fork the call).

Policy System Layer. The policy *database* contains relatively static information such as the mapping between network terms and policy terms. The policy *store* contains dynamic information about policies and their activation. Although the same kind of technology could be used for both, it is convenient to use a conventional RDBMS for the policy database and an XML database (tuple space server) for the policy store (since policies and policy-related information are represented in XML).

User Interface Layer. The policy *wizard* is used to create and edit policies in a user-friendly fashion. The wizard is web-based and multi-lingual. The *context system* is used

to link presence and availability services to the policy system. For example, this allows policies to be triggered by someone's presence or availability (e.g. as indicated by an active badge system or a computerised appointment system).

3. Policy Server Operation

We illustrate the operation of the policy server through of a worked example. Initially, we present a policy written in APPEL, an XML-based policy language developed as part of the ACCENT project, and explain the relevant features of this language as we go along. Complete details of the language can be found in [Reiff-Marganiec04c].

3.1. A First Policy

Suppose that `lb` is a user of the policy server and has a single call control policy, named `fwdLateCallsVM`, stating: from 10th to 14th January 2005, user `lb` would prefer all incoming calls received after 3pm to be forwarded to voicemail.

The basic action associated with this call control policy is forwarding of a call to voicemail – clearly achievable without policies. However, this simple example illustrates some of the power and flexibility associated with policies. Policies are *user*-defined not *network*-defined, so the user is not limited to a generic network implementation. Policies permit fine-grained and flexible control over calls, allowing exactly what the user requires (generally not possible with traditional features or services). In this example, the user specifies a *valid from* and *valid to* date for the policy, and chooses only to apply the policy after 15:00 during this range of dates. Significantly, policies may have a *preference*. In the example above, the use of `prefer` indicates the strength of feeling associated with this policy. As will be discussed later in the paper, this provides us with one basis for the resolution of conflicts. The above policy can be expressed in APPEL as follows:

```
    <policy owner="lb@cs.stir.ac.uk" applies_to="lb@cs.stir.ac.uk"
id="fwdLateCallsVM" enabled="true" changed="2004-12-22T16:10:00"
valid_from="2005-01-10T00:00:00" valid_to="2005-01-14T23:59:00">
     <preference>prefer</preference>
     <policy_rule>
          <trigger>connect_incoming</trigger>
          <condition>
               <parameter>time</parameter>
               <operator>gt</operator>
               <value>15:00</value>
          </condition>
          <action arg1=":voicemail">forward_to(arg1)</action>
     </policy_rule>
    </policy>
```

The *policy* tag specifies the attributes such as the *owner* of the policy, who the policy should *apply to*, a unique user *identifier* for the policy, whether is it currently *enabled* and a timestamp (in XML schema format) to identify when the policy was last *changed*. Also included here are the optional attributes `valid_from` and `valid_to`, stating how long a policy should remain valid.

The second tag states a *preference* level that is to be applied to the policy. A range of preferences may be applied as follows (in decreasing order of strength of feeling):

must, should, prefer. There are also negative forms of these preferences: must_not, should_not, prefer_not. Alternatively, a policy writer may choose to *omit* a preference, meaning indifference about how strongly the policy should be enforced.

The third tag states the *policy rule* itself. Our rules follow a traditional event-condition-action (ECA) style, but we refer to the triggering events of the policy simply as *triggers*. In order for a policy *action* to be fired, a policy's triggers will need to match the triggers associated with the current call, as well as having *conditions* that hold true. Policy variables (with a colon-prefixed name) may be defined independently of a policy, typically through the policy wizard interface. This allows general policies to be defined, but be specialised as required. In the policy above, the variable :voicemail will be separately set to a particular voicemail address.

The process of selecting and applying policies is discussed in section 3.3 below. The range of vocabulary for the component parts of a policy rule is given in table 1.

Table 1. APPEL vocabulary for call control

Element	Call Control
Trigger	absent(*address*), available(*address*), bandwidth_request, connect, connect_incoming, connect_outgoing, disconnect, disconnect_incoming, disconnect_outgoing, event, no_answer(*period*), no_answer_incoming(*period*), no_answer_outgoing(*period*), present(*address*), register, register_ incoming, register_outgoing, unavailable(*address*)
Condition	active_content, bandwidth, call_content, call_type, callee, caller, capability, capability_set, cost, date, destination_address, device, location, medium, network_type, priority, quality, role, signalling_ address, source_address, time, topic, traffic_load
Action	add_caller(*method*), add_medium(*medium*), add_party(*address*), confirm_bandwidth, connect_to(*address*), fork_to(*address*), forward_to(*address*), log_call(*URL*), note_availability(*topic*), note_presence(*location*), play_clip(*URL*), reject_call(*reason*), reject_bandwidth(*limit*), remove_medium(*medium*), remove_party(*address*), send_message(*URL,message*)

3.2. An Example SIP Communication

It is the role of a SIP proxy module to intercept messages at the SIP proxy server, and to process them by extracting key information as a list of variable-value pairs. This information is then passed to the policy server.

Suppose we have two users: kjt@abc.com and lb@cs.stir.ac.uk and that user kjt@abc.com places a call to lb@cs.stir.ac.uk. This call is initially passed to the policy server associated with user kjt (as an outgoing call). Suppose that at this stage the communication server informs the policy server that the call can be tagged as having been made from a PDA, and has been identified as being a long-distance call.

The call is then passed to any further policy servers en route, and finally reaches the policy server associated with the end user lb (as an incoming call). On intercepting this message, the terminating SIP proxy module extracts all relevant information and passes it to the policy server. The policy server determines which protocol is being

used, in this case SIP, and then consults a terminology mapping database in order to translate protocol-specific terminology into common terms used by the policy server.

The policy server stores the extracted information, now in a protocol-independent format, in a hashtable (known as the environmental hashtable). This information consists of a set of field–value pairs as follows (dir is the call direction):

```
SERVER_NAME          d254196.cs.stir.ac.uk
user                 lb@cs.stir.ac.uk
dir                  in
caller               [kjt@abc.com]
callee               [lb@cs.stir.ac.uk]
triggers             [connect, connect_incoming]
device               [PDA]
call_type            [long-distance]
```

A variety of other fields may exist in the message and can be extracted and stored in the hashtable, e.g. to mirror the different terms that may be used as conditions (see table 1).

3.3. Applying Policies

A core task of the policy server is to determine which policies, if any, should be considered with respect to the current call. In our example, we have seen a single policy, `fwdLateCallsVM`, that we can see intuitively should be applied to the above call. The policy server breaks this decision down into four sequential stages discussed further below:

1. consider policies that are *enabled*;
2. determine if a particular policy *applies to* the current call, based on the *user* or *domain* information;
3. determine if a policy is *triggered by* the current call, based on *call* and *environmental* information; if so, the policy's actions are identified;
4. if there is a choice of actions to be taken, apply a *conflict resolution mechanism*.

1. Is a policy *enabled*?

This is straightforward to determine directly by reading the relevant attributes of the policy description. These are the `enabled` attribute, and the (optional) `valid_from` and `valid_to` fields. A further optional attribute states a user's current `profile`. If the user selects a particular profile, e.g. at home, on holiday, or busy, a policy for a profile must match this to be enabled. A policy without a profile is enabled in all cases.

2. Does a policy *apply to* the current call?

To determine if a policy applies to the current call, it is necessary to compare the `applies_to` attribute of the policy's description with the `user` field of the environmental hashtable. Since this latter field was set to `lb@cs.stir.ac.uk`, both values match. Consequently, this policy is said to *apply to* the current call. Note that the policy's `applies_to` attribute may refer to a *domain* instead of an individual user, meaning that the policy should be applied to all users within that domain.

3. Is a policy *triggered by* the current call?

It is necessary here to establish that both the *event* (trigger) and the *condition* hold

before determining whether the *action(s)* should be applied. The policy server must first compare the trigger field(s) of the policy with the trigger(s) identified in the environmental hashtable. In this example, we see that both the policy and the current call refer to an incoming call. With respect to the policy's condition, the policy will be triggered *if* the current time is greater than 15:00. We will suppose here that it is later than 15:00, and hence the policy will be *triggered by* the current call.

4. What are the appropriate policy *action(s)* for the current call?

Finally, the policy server must determine which triggered (and enabled) policies should be applied. If no policies were found to be triggered, the policy server needs to take no further action for the current incoming call. The policy server intervenes no further in the call and control is left to the underlying communications layer.

In our example, we have presented a single policy and shown this to be enabled and triggered. In this instance, the policy server's action is simple: the policy `fwdLateCallsVM` will be applied, and its action `forward_to(:voicemail)` will occur. The policy server will issue a message to this effect to the underlying communication layer, via the SIP proxy module, in this case instructing the SIP protocol to forward the current call.

Identifying the appropriate action will unfortunately not always be so straightforward. It is possible that a user will have a number of policies that are all triggered for a particular call, some of which may conflict with one another. In such a case, the policy server analyses all possible policies and selects the most appropriate resolution.

Having presented an overview of how the policy system processes a policy in the context of a call, the rest of the paper will now focus on the last stage of this process, handling the resolution of conflicts.

4. Handling Conflict on a Single Policy Server

There are two very distinct situations in which call control policies may conflict. The first is in conflicts between policies known to a single policy server. The second is the more complex case of policies in a distributed environment, i.e. where conflicts occur between policies associated with two or more policy servers. This section addresses the simpler 'localised' case; section 5 will progress to issues of conflict in a distributed setting.

In order to illustrate how conflicts are handled by the policy server, we present a second simple policy that conflicts with the first policy (seen in section 3.1).

4.1. A Second (Conflicting) Policy

This second policy is intentionally similar in style to the first policy, since the focus here is on the conflict handling mechanism rather than the range of policies expressible using APPEL. A variety of policy examples can be found in [Huang05] [Reiff-Marganiec04c].

Suppose that user `lb` specifies a policy, `fwdLDCallsHome`, stating: all long-distance evening calls after 6pm must be forwarded to `lb`'s home address.

This is expressible in APPEL as follows:

```
<policy owner="lb@cs.stir.ac.uk" applies_to="lb@cs.stir.ac.uk"
id="fwdLDCallsHome" enabled="true" changed="2004-12-22T16:15:00">
  <preference>must</preference>
  <policy_rule>
    <trigger>connect_incoming</trigger>
    <conditions>
        <and/>
        <condition>
            <parameter>call-type</parameter>
            <operator>eq</operator>
            <value>long-distance</value>
        </condition>
        <condition>
            <parameter>time</parameter>
            <operator>gt</operator>
            <value>18:00</value>
        </condition>
    </conditions>
    <action arg1="lb@lbhome.org.uk">forward_to(arg1)</action>
  </policy_rule>
</policy>
```

In this policy, we have an unrestricted period of validity for the policy (since the `valid_from` and `valid_to` attributes have been omitted). The preference level has been stated as `must`, indicating that the user feels very strongly that, if triggered, this policy must be applied. Whilst the triggering condition (`connect_incoming`) is identical to our original policy, an extra condition has been included: in addition to a condition guarding the time, the policy also checks whether the call type has been tagged as being long distance. Finally, both policies specify a forwarding action, with the destination target differing in each.

If `lb` receives a long-distance incoming call after 18:00, both `fwdLateCallsVM` and `fwdLDCallsHome` are *enabled* and *apply to* the call (w.r.t. the user). Furthermore both policies are *triggered by* the call, i.e. the triggers and conditions hold for each policy.

Consequently, the policy server has a choice of actions: `forward_to(:voicemail)` and `forward_to(lb@lbhome.org.uk)`. In general, a policy server may ask the communications layer to apply multiple actions to the call if they do not conflict with one another, e.g. adding an additional party to a call and also adding a video channel. Note also that a policy server will filter duplicate actions, should they be encountered, to ensure that a given action is only carried out once regardless of how many policies request it.

Returning to our example, the two actions we encountered above can be seen as conflicting, since it is only possible to forward to a single destination (alternative functionality that allows forwarding to multiple destinations is provided by the action `fork_to(`*`address`*`)`). Determining and handling such conflicts is the role of the policy server's conflict resolution handler, as discussed in the following section.

4.2. Conflict Resolution Handling

The key design decision in conflict handling is whether the detection and resolution processes should be internalised, with functionality embedded within the policy server, or whether it should be externalised, thus allowing user-level control of the resolution process. The latter approach was taken, in keeping with ACCENT's overall aim of

providing a user-oriented and totally flexible approach to call control.

In a similar way to which policies are used to specify how a user can achieve control over a call, *resolution policies* can be specified by a user (or system administrator) to state under what conditions, and in what way, a policy server should react to conflicts.

The specification of resolution policies

A characteristic of the extensible nature of our policy language, APPEL, is that *resolution policies* can be specified using the same basic language as our call control policies. The structure of both types of policy is identical, the only difference being in the vocabulary used. Even here, much vocabulary is shared, although it may be used in a different context. For example, resolution policy *triggers* share a vocabulary with policy *actions*: a particular combination of call control policy actions will become a resolution policy's trigger.

Resolution policies are distinguished from call control policies by the use of a different tag, namely a `<resolution>` tag in place of a `<policy>` tag.

As with all event-condition-action style rules, resolution policies specify trigger(s) and condition(s) under which the specified action(s) should occur. In this case, the triggers and conditions identify a situation that the policy server will consider as a conflict, and the policy action provides the resolution to be enforced.

Various types of action are possible within the resolution process; these can be categorised as being one of two types, either a *regular* action or a *relative* action.

Regular actions versus relative actions

Regular actions are the simplest of these two styles of action, since they require no further processing on behalf of the policy server and require no new vocabulary. Such actions share the vocabulary of triggers (and hence call control policy actions). For example, a resolution policy may request one or more of the actions `add_medium(`*`medium`*`)`, `forward_to(`*`address`*`)` or `reject_call(`*`reason`*`)`. Note that the resolution action need not necessarily match one of the resolution triggers.

In contrast, *relative actions* require a little more processing by the policy server and require a small extension to our vocabulary. An obvious example here is in processing the *preferences* that have been specified for our regular policies above. So, in the case of our two conflicting forward events, we wish to choose the policy with the highest preference level. To achieve this we can specify a resolution policy that, under specified triggers and conditions, will perform the relative action: `apply_preference`. An example of this, incorporated into a full policy in APPEL, is provided below.

Other relative actions implemented so far include specifying whether the resolution should apply a policy with a *negative*/*positive preference*, or one associated with the *caller*/*callee*, or the *newest*/*oldest* policy, or simply even the *first* policy encountered (see also below). These actions are all prefixed by the keyword `apply` (to clearly distinguish them from regular actions). This is followed by one of the following keywords: neg_preference, pos_preference, caller, callee, newest, oldest, first. We have also implemented an additional default resolution scheme, `apply_default`, that steps through a sequence of pre-defined relative actions in order to find a resolution. Should this sequence still fail to find a resolution, `apply_default` selects the *first* policy encountered to be the resolution.

Applying resolution policies

The process of applying resolution policies is almost identical to the process of applying call control policies, as described in the four steps of section 3.3.

In order to handle *resolution policies*, the fact that they are identical in format to the regular policies allows us to reuse exactly the same first three steps (omitting only what would be a recursive call to the fourth resolution step). The only difference is in the parameterisation of the three processes: instead of evaluating the normal policies with respect to the call information (information stored in the environmental hashtable), resolution policies are evaluated with respect to the list of *actions* waiting to be resolved. The resolution process is thus as follows:

1. consider resolution policies that are *enabled*;
2. determine if the resolution policy *applies to* the conflicting policies, based on the *user* or *domain* information;
3. determine if the resolution policy is *triggered by* the conflicting policies, based on *policy* and *environment* information.

Both types of policy may make use of environment information e.g. time of day or user availability.

An example resolution policy

The following policy, `noMultipleFwds`, is an APPEL resolution policy for the conflict described above. It applies to everyone in the `cs.stir.ac.uk` domain. The policy provides a resolution for the occurrence of two forwarding events where the forwarding targets are different. More specifically, it is also necessary to consider if the preferences (associated with each event) have opposite signs, that is if one preference is negative (e.g. must_not) and the other preference is positive (e.g. should). These cases are illustrated in the following table:

Table 2. Combinations of forwarding destinations and preferences that cause conflict

Forwarding Destination	Preferences	Conflict or no conflict
Different	Opposite signs	No conflict - e.g. must_not fwd(A) and should fwd(B)
Different	Same signs	**Conflict** - e.g. must fwd(A) and prefer fwd(B)
Same	Opposite signs	**Conflict** - e.g. must_not fwd(A) and prefer fwd(A)
Same	Same signs	No conflict - e.g. should fwd(A) and prefer fwd(A)

There are two rows in this table that indicate conflicting conditions; these are folded into a single resolution policy using a boolean combination of conditions as follows:

```
    <resolution owner="lb@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk"
id="noMultipleFwds" enabled="true" changed="2004-12-22T16:20:00">
     <policy_rule>
          <triggers>
               <and/>
               <trigger arg1="VAR1">forward_to(arg1)</trigger>
               <trigger arg2="VAR2">forward_to(arg2)</trigger>
          </triggers>
          <conditions>
               <or/>    <!-- rows 2/ 3 condition give conflict -->
```

```
            <conditions>
                <and/>  <!-- different destination, same signs -->
                <condition>
                    <parameter>VAR1</parameter>
                    <operator>ne</operator>
                    <parameter>VAR2</parameter>
                </condition>
                <condition>
                    <parameter>preferences</parameter>
                    <operator>ne</operator>
                    <value>opposites</value>
                </condition>
            </conditions>
            <conditions>
                <and/>  <!-- same destination, opposite signs -->
                <condition>
                    <parameter>VAR1</parameter>
                    <operator>eq</operator>
                    <parameter>VAR2</parameter>
                </condition>
                <condition>
                    <parameter>preferences</parameter>
                    <operator>eq</operator>
                    <value>opposites</value>
                </condition>
            </conditions>
        </conditions>
        <actions>
            <orelse/>
            <action>apply_preference</action>
            <action>apply_newest</action>
        </actions>
    </policy_rule>
</resolution>
```

This policy's triggers are two `forward_to` actions. One point to note here is the use of *variables* in the parameters of actions, e.g. `forward_to(VAR1)`. This feature enhances the expressibility of the resolution policies, allowing the variables to be bound to real values at run-time, i.e. identifying the target destination as the resolution policies are actually evaluated. In this example, the two variables `VAR1` and `VAR2` are bound at run-time to the voicemail address and to `lb@lbhome.org.uk` respectively.

The resolution policy's *conditions* checks whether:

- the targets of these two forward actions are different and the preferences have the same sign, i.e. that a conflict exists in simultaneously trying to forward to two different destinations, *or*
- the targets of these two forward actions are the same and the preferences have opposite signs, i.e. that a conflict exists between a negative preference and a positive preference for the same forwarding action.

In our example, the values of the two variables are different (the voicemail address is different to `lb@lbhome.org.uk`) and the preferences are both positive, i.e. have the same sign (`prefer` and `must`). Therefore, the first condition holds.

The policy's resolution *action* combines two alternative actions with the `orelse` operator. This operator ensures that the first action is tried first, i.e. the *preference* associated with the two policies should be compared. If this fails to provide a resolution, then the second action should be considered, i.e. the timestamps of the two

policies should be compared and the *newest* one selected.

In the case of the resolution policy still being unable to draw a distinction between the call control policies, as may occur with the use of relative resolution actions, *generic resolution policies* may be applied, as will be discussed below. If this still fails to resolve a conflict, the actual action to be taken will be determined by the policy server implementation: a method is called that corresponds to the `apply_default` relative action (described above) to ensure a resolution is reached.

For the example above, it is sufficient to choose the first relative action, i.e. resolving conflict on the basis of the policy preferences. The first call control policy, `fwdLateCallsVM`, has preference level `prefer` whilst the second call control policy, `fwdLDCallsHome`, has preference level `must`. In this instance, the first policy will obviously lose out to the stronger preference of the second policy. Consequently, the action associated with the second policy will be selected and the policy server will notify the communications layer to apply `forward_to(lb@lbhome.org.uk)` to the current incoming call.

If a situation arose involving a further conflicting action, specified with a `must` preference and a *timestamp* of 2004-12-22T16:05:00, the second resolution action, `apply_newest` would come into play. Having determined that the first action, `apply_preference`, could not provide a resolution, the policy server would evaluate the conflicting policy timestamps. This would result in the second policy being chosen.

Specific resolution policies versus generic resolution policies

We refer to the above style of resolution as a *specific resolution policy* since it relates to a specific combination of triggers, namely a conflict between two forwarding events. Our experience in writing such policies has shown that it is not always straightforward to correctly enumerate all required conditions in order to guarantee a resolution is reached.

Precisely because of these difficulties, a set of *generic resolution policies* have also been written, covering a range of possible conflicts and providing suggested resolution action(s). The resolution is, of course, able to be tailored (specialised) by the user as required. The term generic is used here in the sense that these policies do not relate to a specific combination of triggers; there are in fact no explicit triggers for such policies, only conditions and actions. These policies can still be specified using APPEL, by simply omitting the trigger tags. However, the policy server treats generic policies differently to specific resolution policies. In order to be triggered correctly, i.e. not triggered for unrelated actions, generic resolution policies can be viewed as having an *implicit* trigger that checks if the base-name (discounting any parameters) of a call control action matches the base-name of another action, e.g. if there are two `add_medium` actions or two `reject_call` actions. Also, due to their generic nature, the policy server ensures that generic resolution policies are only ever applied *after* all specific resolution policies have been applied.

In our current implementation, all identified generic conflicts have been grouped together into five generic resolution policies. These five sets of conflicts cover the following conditions: preferences are exact opposites, preferences are opposites but not exact opposites, preferences are not opposites, preferences are the same and timestamps are different, and preferences and timestamps are the same.

Finally, it is also worth pointing out that *all* of our resolution policies, whether

specific or generic, can be applied at design time as well as at run-time, although their use at design time has not yet been implemented. At design-time, as soon as the user uploads policies via the policy wizard, the policy server can check for conflicts against a set of provided resolution policies. Should a conflict be detected, any triggered resolution policies will be identified to the user, allowing him/ her to specialise the resolution actions as desired.

5. Handling Conflict in a Distributed Setting

The policy server architecture shown in figure 1 requires multiple policy servers to cooperate in a distributed environment. This presents us with a number of different ways in which we could perform conflict resolution across multiple policy servers.

5.1. An Example of Conflicting Distributed Policies

Suppose we have the following two distributed policies that are triggered for the same call from user `kjt` to user `lb`. The first policy, `noAddPartyOutgoing`, is applied to an *outgoing* call by the caller's policy server and states that under no condition must an additional party be added to the call:

```
    <policy owner="kjt@abc.com" applies_to="kjt@abc.com"
id="noAddPartyOutgoing" enabled="true" changed="2004-12-22T16:25:00">
     <preference>must_not</preference>
     <policy_rule>
          <trigger>connect_outgoing</trigger>
          <action arg1="">add_party(arg1)</action>
     </policy_rule>
    </policy>
```

The second policy, `addPartyHead`, is applied further down the communication chain, at the callee's policy server. This policy states that for *incoming* calls from `kjt`, `lb` would `prefer` an additional party, `head@cs.stir.ac.uk`, to be conferenced into the call:

```
    <policy owner="lb@cs.stir.ac.uk" applies_to="lb@cs.stir.ac.uk"
id="addPartyHead" enabled="true" changed="2004-12-22T16:30:00">
     <preference>prefer</preference>
     <policy_rule>
          <trigger>connect_incoming</trigger>
          <condition>
               <parameter>caller</parameter>
               <operator>eq</operator>
               <value>kjt@abc.com</value>
          </condition>
          <action arg1="head@cs.stir.ac.uk">add_party(arg1)</ action>
     </policy_rule>
    </policy>
```

Clearly these two policies conflict with each other, one preventing another party being added to the call, the other requesting the addition of another party. Glancing at the preferences for the two policies indicates that an easy solution is available here, since the *must not* precedence of the first policy is stronger than the *prefer* precedence of the

second policy. However, automating this in a distributed setting provides a number of challenges.

5.2. Distributed Conflict Resolution

Design issues

Many approaches to the run-time resolution of conflicts make use of a centralised resource, such as a centralised negotiator [Griffeth94], a tuple space [Amer00] or a distinct component such as a feature interaction manager [Jia03] (although communication in the latter approach is via a distributed tuple space).

A major design decision in the development of the resolution technique for the ACCENT project has been to avoid any such centralised approach. This mirrors our actual environment where end-users are clearly distributed across communication servers and subsequently call control and resolution policies are distributed across policy servers. So, rather than having a single interaction manager, all policy servers are capable of performing conflict resolution. Unfortunately, distributed policy servers cannot operate independently of one another, since the policies they enforce are intended to apply end-to-end over a call. An illustration of this will be given below. Therefore some mechanism is needed for distributed communication in order to reach an agreed resolution for conflicting policies in a distributed environment.

One possible solution would be to allow policy servers to communicate directly with one another. (In fact, policy servers have pre/post-negotiation hooks for this purpose.) However, this could fail to preserve the layered architecture of the policy system. The role of the policy server is to intercept and, where necessary, amend messages from the underlying communications layer. It is the role of the communications layer to determine call routing, i.e. deciding which server will be visited next in the communication chain. Providing routing information for policy servers to communicate directly would not be appropriate in a layered architecture.

A further solution would be to employ a distributed tuple space for communication, as in the approach of [Jia03]. However, our architecture already has a (non-distributed) tuple space associated with each policy server, acting as a policy store. Consequently, our solution to distributed conflict resolution takes advantage of this fact: one of the tuple spaces along the call's route, e.g. the tuple space located at the originating end, acts as a temporary store (blackboard) for information relevant to the resolution process for that particular call. This can be considered a decentralised resource. Although a single blackboard is used for the duration of a call, its location will differ on a call-by-call basis.

In summary, our approach:

- avoids a totally centralised resource;
- empowers each policy server to perform conflict resolution;
- uses the tuple space (policy store) associated with a particular policy server as a blackboard for temporary storage of resolution information for the duration of a call; such blackboards are established on a call-by-call basis.

Problems with sequential (independent) resolutions

Before detailing our chosen approach, we believe it is worth explaining problems that would arise should a sequence of independent resolutions be applied across the

communication chain. The reasons why this approach is not suitable become relevant later in the discussion.

With this approach, on intercepting a call each policy server would perform its own conflict resolution independently of any other policy servers. This would result in a set of policies being enforced for the call at its current local location.

In the example above, the caller's policy server would enforce the `noAddPartyOutgoing` policy. Since this is a negative policy, and the policy server has no other requests to perform any other actions, just passing control back to the underlying communications layer actually enforces this policy locally: no party is added. On reaching the destination, however, the second policy, `addPartyHead`, would request that an additional party is added; this would be enforced since there are no conflicting policies locally.

In this case, it could be argued that the initial policy has been applied too soon in the communication chain and should have been deferred, being considered in conjunction with the second policy at the destination policy server. Had this been the case, a simple resolution mechanism based on *preferences* would have ensured the appropriate resolution.

The problem of employing policies too soon is not restricted to this particular example. Forwarding, forking and blocking actions are particularly problematic here, since they alter the route of a communication. For example, if policy servers independently enforce forwarding actions (locally) en route, the route of the call may be altered by the first such policy; subsequent policy servers from the original route may never be reached. Thus there is never any chance for a consolidated global resolution process. Instead there is an implicit priority of policies depending on the policy server's location in the communication chain.

Clearly, therefore, there needs to be distributed communication in order to reach an agreed resolution for conflicting policies in a distributed environment.

Achieving distributed communication

In our approach, the policy store associated with the first policy server with a triggered policy assumes the role of blackboard for the duration of the call. Policy servers make use of this resource as follows:

- On intercepting a call, the *caller's* policy server determines which policies are enabled and are triggered by the call. Rather than resolving any conflicts directly, all possible actions are deferred, being written to and accumulated on the blackboard. All policies are labelled with a unique call identifier (e.g. the SIP call identifier). All enabled *resolution policies* from that policy server are also written to this store, again identified by the call identifier. Finally, the call data itself is modified to carry a reference to the caller's tuple space. This ensures that all *subsequent* policy servers can access and store resolution information on the blackboard that relates to the current call. The mechanism for carrying the blackboard reference depends on the communications layer. In SIP or H.323, for example, it is carried as an extension header field.
- On reaching the *callee's* policy server, the server consults local policies and identifies those that are enabled and are triggered by the call. This policy server consults all call control policies from the blackboard, and a list of possible actions is generated. The server starts the resolution process by analyzing this set of accumulated policy actions. In the case of any conflicts,

all accumulated resolution policies are considered. The chosen resolution is then enforced by passing relevant information to the communication layer via the SIP proxy module as discussed before. Note that this process takes place at the callee's policy server, i.e. before the call actually reaches the end user. Only if a resolution can be reached will the call be established and the policies activated.

5.3. Discussion: Issues Associated with Distributed Conflict Resolution

A number of issues remain with the distributed resolution of conflicts:

Immediate versus deferred actions

The issue of choosing between *immediate* actions, a result of an independent (localised) resolution approach, versus *deferred* actions, as occurs with the proposed blackboard approach, is unfortunately not straightforward. We have already given a justification of avoiding immediate actions on the grounds of preventing the implicit priority of policies based on the policy server's location in the communication chain. However, the deferral of actions also causes potential problems.

As mentioned earlier, policy actions involving forwarding, forking or blocking are particularly problematic in that they affect the route of communication. As a consequence, they also potentially influence the chain of policy servers visited and hence the policies encountered en route. If such actions are deferred to the destination of the call, suppose the policy action selected by the resolution process involves forwarding (or forking) to a new location. This may have the effect of visiting one or more new policy server(s) and applying new policies to the call. There is clearly no way that these policies can be included in the first round of negotiations, hence a second round of negotiation must take place when this new location is reached. Obviously this could theoretically occur repeatedly depending on the policy action selected at the new location.

A rather complex solution could make use of the forking functionality offered by the communications layer, to offer an 'anticipatory' style of resolution strategy. On encountering a potential forwarding action, both the original route and the forwarding route could be tentatively followed, i.e. without committing to a particular communication route at this stage. Note that policies stored on the blackboard after any such forking would need to be distinguished by means of a suffix on the call identifier. Since multiple 'end' policy servers are now contacted, due to the earlier forwarding action, the resolution module must decide which is the most appropriate destination device based on the accumulated information. However, one further complication remains concerning which of the end policy servers should actually instigate the resolution process. The blackboard can again be used as a common resource to allow policy servers to make this decision; it simply adds further complexity to an already complex solution.

A similar process could be followed on encountering a potential forking action. However a key difference exists between this and the forwarding process above. With traditional forking, each of the end devices is contacted: the first one to answer will be connected. This implies that each 'end' policy server should perform resolution for its own communication route, and will access the blackboard in order to analyse the accumulated policies for the route in question. If the policies do not prevent the

communication, the end device of each route is contacted. In this way, the resolution process is acting as a filter, where the final 'resolution' is determined in favour of the device answered first.

Increasing the complexity of the resolution process to take into account the above complications has the advantage of avoiding taking any actions that, with a global view of the system (and/ or the benefit of foresight), could be deemed to be premature. However, the clear disadvantage is that the call setup process is significantly complicated, potentially unnecessarily (e.g. if no future conflict actually exists and the action could be taken immediately with no future adverse effect).

Triggers that rely on environmental information

The solution that we have proposed above (section 5.2) concludes with the recognition that conflict resolution occurs at the 'end' policy server, i.e. before the end device is actually contacted. A drawback of this approach concerns policies (specifically triggers) that rely on certain types of environment information. For example, the APPEL language contains a `no_answer` trigger, designed to allow a policy to specify that a subsequent action can be attempted should a phone remain unanswered for a particular number of rings. However, this information is only obtainable if the end device is actually contacted.

It would be possible to follow a similar 'anticipatory' forking strategy as described above in order to explore potential conflicts should this route be followed. However, as above, potential benefits are not deemed sufficient to justify the added complexity.

6. Conclusions

This paper has explored the use of policies for the provision of user-oriented and flexible control of calls in a telecommunications domain. The potential benefits of policies are great, with policies able to:

- replace (or at least complement) the more traditional features and services as mechanisms for achieving call control;
- be arranged hierarchically, to allow the specification of policies for individual users, their organisations and their service providers as well;
- be specified as having distributed effect, e.g. permitting a caller's policy to provide some level of control as a call arrives at its destination.

The ACCENT policy system architecture has been presented, showing how policies are handled by policy servers. These intercept, interpret and (as necessary) alter call messages in the communications layer. The issue of resolving conflicts that arise between policies has been discussed and illustrated through a number of examples.

Our approach has a number of distinctive advantages. The APPEL language allows the use of ECA-style policies for call control, with preferences supporting resolution. A policy wizard allows non-technical users to create and amend policies in a user-friendly manner. Specialised policies, also written in APPEL, allow flexible control of the resolution process. A distinction is drawn between two possible styles of resolution action, namely regular and relative actions, and also between specific and generic resolution policies. A policy store's tuple space doubles as a blackboard (established on a call-by-call basis) during the resolution process.

Importantly, the issue of achieving resolution in a truly distributed setting (in the

absence of a global view of policies) has been shown to be far from trivial. Several research issues remain. Although our proposed approach suffers from a number of limitations, alternative solutions involve a significant step up in complexity (and hence inefficiency) for relatively little gain.

Finally, the issue of the performance of our approach has still to be fully investigated. The implementation of the policy server architecture and the basic resolution mechanism is now largely complete. Early indications suggest that the delay imposed in the call setup process will fall within an acceptable range. However, work still remains in evaluating the performance of our architecture in the more difficult cases described in section 5, and also in evaluating the system when fully connected in a distributed environment.

Acknowledgements

The work reported in this paper was financially supported by the Engineering and Physical Sciences Research Council (under grant GR/R31263) and by Mitel Networks Corporation. Stephan Reiff-Marganiec (originally University of Stirling, now University of Leicester) designed and implemented the base policy system. Jianxiong Pang (University of Stirling) integrated policy support with the Mitel 7000 ICS. The authors warmly thank Tom Gray (Mitel), Peter Perry (Mitel) and Joe Ireland (MKC Networks) for their continued support and technical advice.

References

[Amer00] M. Amer, A. Karmouch, T. Gray, S. Mankovskii, "An Agent Model for the Resolution of Feature Conflicts in Telephony", In Journal of Network and Systems Management, 8(3), 2000.

[Amyot03] D. Amyot, L. Logrippo (eds), "Feature Interactions in Telecommunications and Software Systems VII", Ottawa, Canada, IOS Press, Amsterdam, 2003.

[Bertino02] D. Bertino, M. Cochinwala and M. Mesiti, "UCS-Router: A policy engine for enforcing message routing rules in a universal communication system", Proceedings Mobile Data Management, January 2002.

[Calder00] M. Calder, E. Magill (eds), "Feature Interactions in Telecommunications and Software Systems VI", Glasgow, Scotland, IOS Press, Amsterdam, 2000.

[Calder03] M. Calder, M. Kolberg, E.H. Magill, S. Reiff-Marganiec, "Feature Interaction: A Critical Review and Considered Forecast", Computer Networks, 41(1), pp 115-141, January 2003.

[Chentouf03] Z. Chentouf, S. Cherkaoui, A. Khoumsi, "Experimenting with Feature Interaction Management in a SIP Environment", Journal of Telecommunication Systems: Modelling, Analysis, Design and Management, Vol. 24, Kluwer Academic Publishers, 2003.

[Dini04] P. Dini, A. Clemm, T. Gray, F.J. Lin, L. Logrippo, S. Reiff-Marganiec, "Policy-enabled Mechanisms for Feature Interactions: Reality, Expectations, Challenges", Computer Networks, Vol. 45, pp 585-603, Elsevier, 2004.

[Griffeth94] N. Griffeth, H. Velthuijsen, "The Negotiating Agents Approach to Run-time Feature Interaction Resolution", In Feature Interactions in Telecommunications Systems, L.G. Bouma, H. Velthuijsen (eds), IOS Press, Amsterdam, May 1994.

[Huang05] T. Huang, K.J. Turner, "Policy Support for H.323 Call Handling", Computer Standards and Interfaces, February 2005 (in press).

[ITU00] Packet-Based Multimedia Communication Systems, ITU-T H.323, International Telecommunications Union, Geneva, Switzerland, November 2000.

[Jia03] Y. Jia, J.M. Atlee, "Run-time Management of Feature Interactions", Proceedings 6th ICSE Workshop on Component Based Software Engineering: Automated Reasoning and Prediction, I. Crnkovic, H. Schmidt, J. Stafford, K. Wallnau (eds), Portland, Oregon, 2003.

[Keck98] D.O. Keck, P.J. Kühn, "The Feature and Service Interaction Problem in Telecommunications Systems: A Survey", IEEE Transactions on Software Engineering", Vol. 24, pp 779-796, 1998.

[Lupu99] E.C. Lupu, M. Sloman, "Conflicts in Policy-Based Distributed Systems Management", IEEE Transactions on Software Engineering, Vol. 25, No. 6, 1999.

[Mitel04] Mitel 7000 ICS (Integrated Communications Server) Technical Guides, Mitel Networks Corporation, September 2004.

[Nakamura03] M. Nakamura, P. Leelaprute, K. Matsumoto, T. Kikuno, "Detecting Script-to-Script Interactions in Call Processing Language", In D. Amyot and L. Logrippo (eds), Proceedings 7th Feature Interactions in Telecommunications and Software Systems, pp. 215-230, IOS Press, Amsterdam, June 2003.

[Nyholm02] C. Nyholm, "Product Line Development – an Overview", Extended Report for "Building Reliable Component-Based Systems", I. Crnkovic, M. Larsson (eds), Artech House, July 2002. ISBN 1-58053-327-2.

[Reiff-Marganiec02] S. Reiff-Marganiec, K.J. Turner, "Use of Logic to Describe Enhanced Communications Services", In D.A. Peled and M.Y. Vardi (eds), Proceedings Formal Techniques for Networked and Distributed Systems, LNCS 2529, pp. 130–145, Springer, Berlin, Nov. 2002.

[Reiff-Marganiec03] S. Reiff-Marganiec, K.J. Turner, "A Policy Architecture for Enhancing and Controlling Features", In D. Amyot and L. Logrippo (eds), Proceedings 7th Feature Interactions in Telecommunications and Software Systems, pp. 239–246, IOS Press, Amsterdam, June 2003.

[Reiff-Marganiec04a] S. Reiff-Marganiec, "Policies: Giving User Control over Calls", In M.D. Ryan, J.-J. C. Meyer, and H.-D. Ehrich (eds), Objects, Agents and Features, LNCS 2975, pp. 189–208, Springer, Berlin, May 2004.

[Reiff-Marganiec04b] S. Reiff-Marganiec, K.J. Turner, "The ACCENT Policy Server", Technical Report CSM-164, Computing Science and Mathematics, University of Stirling, UK, August 2004.

[Reiff-Marganiec04c] S. Reiff-Marganiec, K.J. Turner, "APPEL: The ACCENT Project Policy Environment/ Language", Technical Report CSM-161, Computing Science and Mathematics, University of Stirling, UK, August 2004.

[Reiff-Marganiec04d] S. Reiff-Marganiec, K.J. Turner, "Feature Interaction in Policies", Computer Networks, 45(5):569–584, August 2004.

[Robinson03] W.N. Robinson, S.D. Pawlowski, V. Volkov, "Requirements Interaction Management", ACM Computing Surveys (CSUR), Vol. 35, Issue 2, pp 132-190, June 2003.

[Rosenberg02] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnson, J. Peterson, R. Sparks, M. Handley, E. Schooler (eds), "SIP: Session Initiation Protocol", RFC 3261, The Internet Society, New York, USA, June 2002.

[Turner04] K.J. Turner, "The ACCENT Policy Wizard", Technical Report CSM-166, Computing Science and Mathematics, University of Stirling, UK, August 2004.

[vanLamsweerde98] A. van Lamsweerde, R. Darimont, E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", IEEE Transactions on Software Engineering", Vol. 24, pp 908-926, 1998.