Gottfried Wilhelm
Leibniz Universität Hannover
Information Systems Institute
Knowledge Based Systems
Prof. Dr. techn. Dipl.-Ing. W. Nejdl
Appelstr. 4
30167 Hannover
Germany

Controlling Dynamic Content Delivery With Policy-Driven Negotiations

by Sebastian Wittler

ID 2070106

University of Hannover, Germany

sebastian_wittler@gmx.de

Master-Thesis

26.09.2006-26.03.2007

First Examiner: Prof. Dr. techn. Dipl.-Ing. Wolfgang Nejdl (KBS, L3S)

Second Examiner: Prof. Dr. Kurt Schneider (SE)

Supervisor: Dipl.-Inf. Daniel Olmedilla (KBS, L3S)

Declaration

I declare and assure that I have written this Master-Thesis by myself, without any help from other persons.

Further, I declare and assure that I have only used the sources and resources that are listed in the last chapter.

Sebastian Wittler

Table of Contents

1 Introduction..........5
2 Policies & Policy-Driven Negotiation..........6
2.1 Motivation..........6
2.2 Trust Negotiation..........9
2.2.1 Credentials..........10
2.2.2 Policies..........11
2.2.3 Examples..........12
3 Challenge: How to protect my Web Resources..........15
3.1 Motivating Scenario..........15
3.2 Challenges..........18
3.2.1 Policy Management Tool..........18
3.2.2 Policy-Based Dynamic Content Generation..........19
3.2.3 Negotiating on the Web..........21
4 Policy Management..........23
4.1 Policies..........23
4.1.1 Assigning policies to resources..........23
4.1.1.1 Example..........25
4.2 The Policy Management Tool..........26
4.2.1 Description..........26
4.2.1.1 The Resource View..........27
4.2.1.2 The Policies View..........28
4.2.1.3 The Policy Editor..........29
4.2.1.4 The Console View..........30
4.2.1.5 The Inherited Policies View..........30
4.2.1.6 RDF Export and Import..........31
4.3 Implementation..........35
4.3.1 Rich Client Platform..........35
4.3.2 Model Implementation..........36
4.3.2.1 Retrieving Policies for a Resource..........37
4.3.2.2 Example..........40
4.3.3 GUI Implementation..........42
4.3.4 Logic Implementation..........46
5 Policy-Driven Protection & Dynamic Content..........47
5.1 Static Protection..........48
5.1.1 Example..........50
5.2 Dynamic Protection..........51
5.2.1 Java Server Pages..........52
5.2.1.1 Custom Tags..........52
5.2.1.2 The policycondition Tag..........53
5.2.1.3 Example..........54
5.3 Implementation..........56
6 Policy-Driven Negotiations on the Web..........58
6.1 The Client..........58
6.1.1 Implementation..........59
6.1.2 Applets..........59

6.1.3 Java Script........60
6.1.4 Applet - Mediator between Client and Server........63
6.1.4.1 Communication Channel........63
6.1.4.2 Waiting Screen........64
6.1.4.3 Displaying the Result........65
6.2 The Server........67
6.2.1 Implementation........68
6.2.1.1 Servlets........68
6.2.1.2 Filters........69
6.2.1.3 Static Protection Filter........70
6.2.1.4 Dynamic Protection Filter........73
7 Related Work........76
8 Conclusions & Open Issues........79
9 References........82

1 Introduction

A new way to get access to sensitive resources in the web is trust negotiation [1]. In contrast to the traditional way of registering and login, among other benefits, it eliminates the requirement to manually register and enter password/login every time. The approach of trust negotiation automatically exchanges policies and credentials (which protect or satisfy a resource) between the involved parties in order to determine if access is granted or not.

While research focuses mainly on trust negotiation in peer to peer networks, this master thesis concentrates on its use and integration in the world wide web and its easy integration into servers and clients. This is necessary, as assigning policies to a resource (file, credential etc.) is not easy (especially for inexperienced users). This thesis suggests and implements approaches to support the user (client and administrators). An easy to use graphical policy management tool and simple, flexible integration of trust negotiation components for administrators was developed. Besides being able to assign policies to whole documents (html, pdf, etc.), referred as static protection), the developed approach also offers the option to protect also content of web pages according to policies (dynamic protection). It uses the Protune/PeerTrust [1,2] framework for negotiation, developed by the L3S Research Center [3] in Hannover.

This thesis provides

- a policy management tool
- a server and client integration of trust negotiation (applet and servlet)
- support for dynamic protection of content according to policies

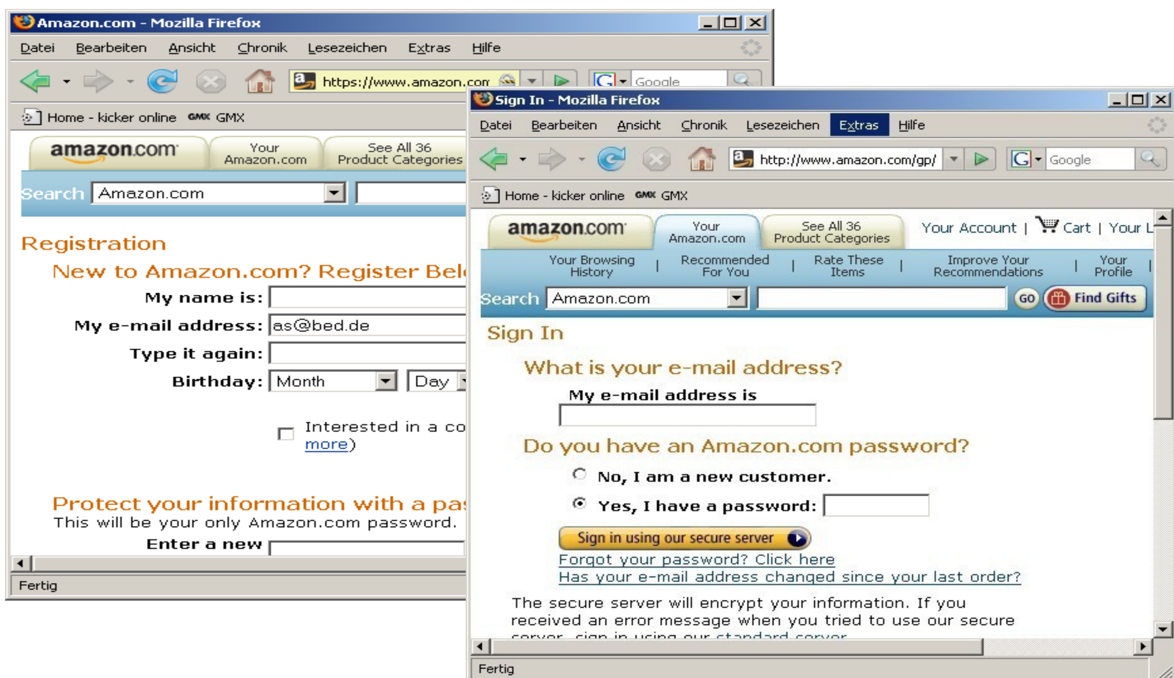
Further information and resources for this thesis can be found in internet under http://www.sebastianwittler.de/master_thesis.

2 Policies & Policy-Driven Negotiation

This chapter will introduce the concept of trust negotiation, which represents an alternative to the traditional registering/login approach predominating in the web, and its main components.

2.1 Motivation

In the Word Wide Web today, most sensitive resources can be accessed via the traditional registering and login mechanism. That means that a user has to fill out a registration form with required data in order to obtain a password and login-name combination with which he can access the desired sensitive resources [1].



This approach is widely used by many companies and services in the World Wide Web (Online shops like Amazon as shown in the screenshots, eMail- and news-services, forums, etc.). Despite of being popular, this technique has some significant disadvantages

- The user has to disclose personal information to the company or service. Although some can be entered voluntary, many fields are required to be filled out. Depending on the site, very private or not obviously relevant information must be entered to obtain access, for instance a credit card number, an address or a telephone number. The customer may not want to disclose these details, but the registration require them. He either may enter all data or skip the registration.
- The customer has often no chance to find out if he can trust the company or service. Credit card numbers, eMail-addresses or phone numbers can be misused. Many criminals in the web try to get customers data (Phishing , spamming, etc.).
- This applies also for the other side, the server. Customers may enter false information, either accidentally or intended (many think its an annoying task and use tools which fill out registration forms automatically or enter nonsense). Verifying customer data is very hard without acknowledgement from trusted third parties.
- The user has to keep all his password/login-name combinations in mind. Cookies may store the necessary data so the user is logged in automatically next time, but cookies can be deleted (by the browser or user).
- Many people do not take care about their passwords. Either they choose ones which can be easily guessed (name of daughter or wife, birthday or simply „password“). Studies have shown that many users are also vulnerable for social engineering (and may so be tricked to reveal their password) [4].

Taking all this into account, the disadvantages lead to the conclusion that the establishment of trust, protection of sensitive information/resources (for both parties) and easy appliance (no need to remember any password any more) would help to improve the traditional registering/login-technique.

The following parts present a possible way which offers all this features and is a popular topic of research (as it is also part of the Semantic Web in which intelligent agents work and collaborate in order to fulfil tasks for its users).

2.2 Trust Negotiation

The desired features stated above (establishment of trust, protection of sensitive information/resources for both parties and ease of use) can be realized in an approach called „Trust Negotiation“ [1].

In contrast to the password/login-paradigm, it fits much closer to the real word. Someone usually trusts a stranger only, if the latter convinces him and if his arguments can be confirmed by trusted third parties, as this diminishes the chance he might lie. If a company searches a new employee for a vacant position, it would not take the statements from the candidates for granted (university grades, skills, foreign languages, etc.), instead it will rely on certificates and credentials from trustworthy third parties (universities, other companies, schools, etc.) which either attest or disprove what the candidates claim. Another example is the need to have a drivers licence to drive a car or to show a passport at the border if someone is visiting another country.

Trust is not normally there from scratch, it develops slowly. If a stranger asks someone to borrow him his car, one would refuse, if someones mother or close friend ask, this will be completely different. So an effort to build trust may take multiple rounds in order to become effective. This really can not be compared to the „one-shot“ trust of traditional register/login-techniques.

As a further prerequisite, trust depends on conditions of everyone. Mark for instance shows his police badge only when he is on duty and asked for it by suspects. Steven on the other hand shows his to women in order to impress them.

So taken all together, verification by trusted third parties, multiple rounds of establishing trust and the dependency of individual conditions are the main concepts of trust negotiation. The main idea is that everybody is only willing to disclose information or resources if the other party fulfil the conditions bound to it.

So if a negotiation between two parties starts, a compromise has to be found (as each side has its own conditions) in order to achieve the goal.

After this short introduction, the next few parts will explain trust negotiation in more detail and provide examples in order to support understanding of the concepts, main ideas and possibilities of this technique.

2.2.1 Credentials

Credentials are statements about someone or something issued by trusted third parties. Examples are university degrees, a driving licence or a police badge. Credentials can be used to establish trust between strangers as already mentioned above, as each one can prove what he says. They play a key role in satisfying the conditions of other persons (if you want to enter into the cinema, show me your ticket – or – if you want to get the job prove me that you are a C# expert). Credentials often have built-in mechanisms (holograms, a photo of the owner, PIN-number etc.) that should protect the often very private content and hinder other people from forging or misusing the credential.

For making trust negotiations on computers possible, credentials have to be digital, of course. As mechanisms like holograms do not work here, a digital counterpart has to be used which also hinders manipulation or misuse of the credential and guarantees that it was really issued by the claimed issuer. Asymmetric (private and public) keys and digital signatures may be used (like in X.509 credentials).

If a user wants to access a sensitive resource from a server, for instance to buy a book in an online shop, and a trust negotiation starts, the server may require the customer to disclose his credit card and customer credential first. Besides digital credentials, conditions like these play also an important part in trust negotiation and are topic of the next section.

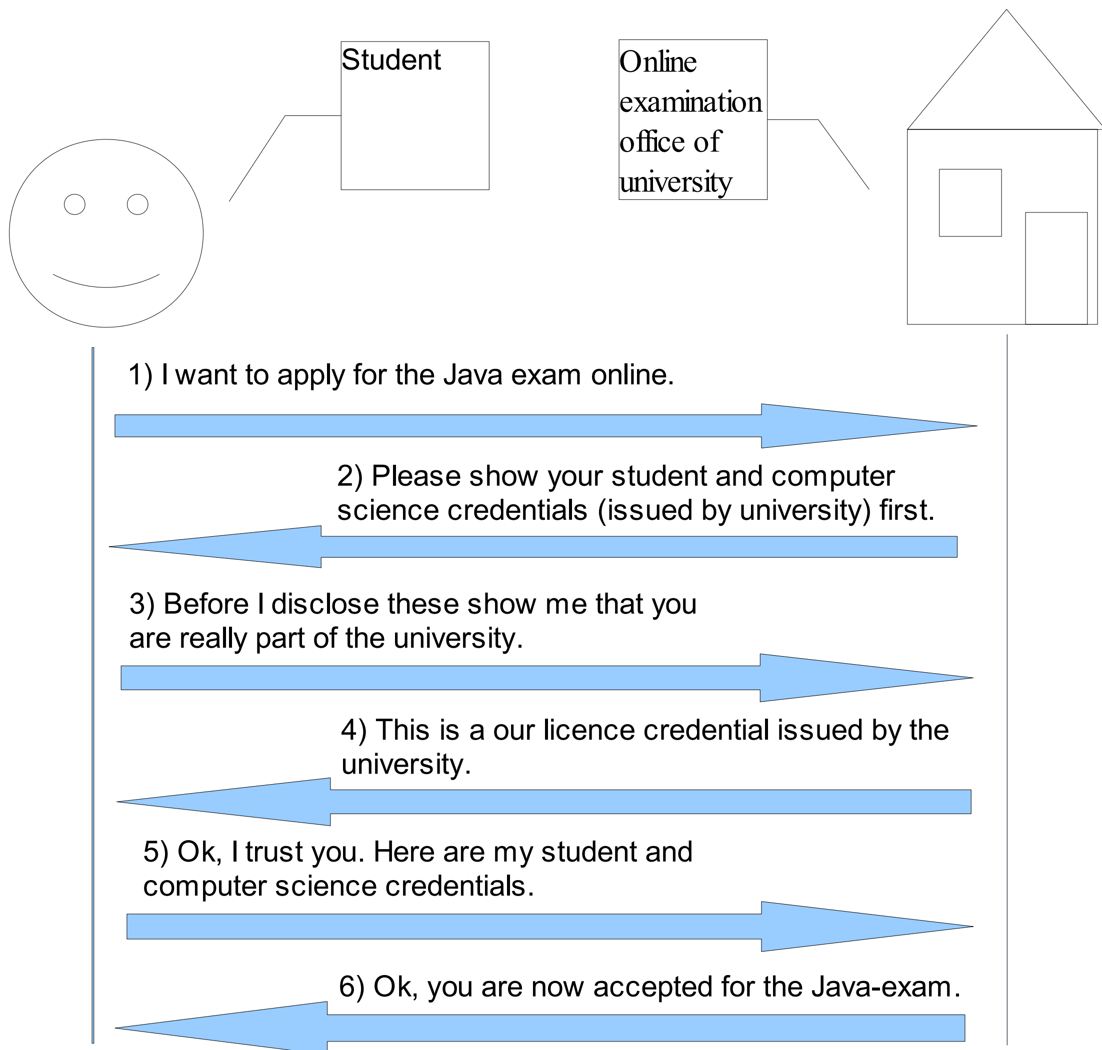
2.2.2 Policies

Specifying conditions like the above example in which the online store requires the customer to disclose his customer and credit card credential is done via policies. They may be assigned to sensitive resources (credentials, files, web pages, etc.) in order to protect them. Another party has to satisfy this policy first before getting access to that resource. This way one can restrict access only to the intended subset of users (the online store from the example only allows his customers who have a credit card to buy something). Everybody should be able to protect his resources via policies, also the customer of the above example who maybe does not want to disclose his credit card credential to everyone.

To help trust negotiation in becoming more and more popular, one aim is to easily enable users to write his own policies. A lot of languages already exist for that (many are logic languages which offer negation, disjunction, conjunction, etc.). A trade-off has to be made between expressiveness and computability. It has also be taken into consideration that users maybe do not know the language and do not want to spend hours on learning it before being able to specify their policy. To solve this problem, the language may be made easier (which maybe can affect expressiveness) or special tools or wizards can assist the user in writing or selecting his policies [5,6].

Policies and credentials are the main ingredients of trust negotiation. Software agents acting for each communication partner automatically exchange the relevant ones in order to establish trust, if succeeded, the requester is granted access to the resource, otherwise not. The difference to the registering/login approach is that each partner may have requirements (policies) for his resources (data, credentials, files, etc.), the negotiation is bidirectional and may last multiple rounds. This greatly helps the user as he does not have to remember and enter passwords any more, the whole negotiation is running in the background. Examples follow which should help understanding the concept of trust negotiation.

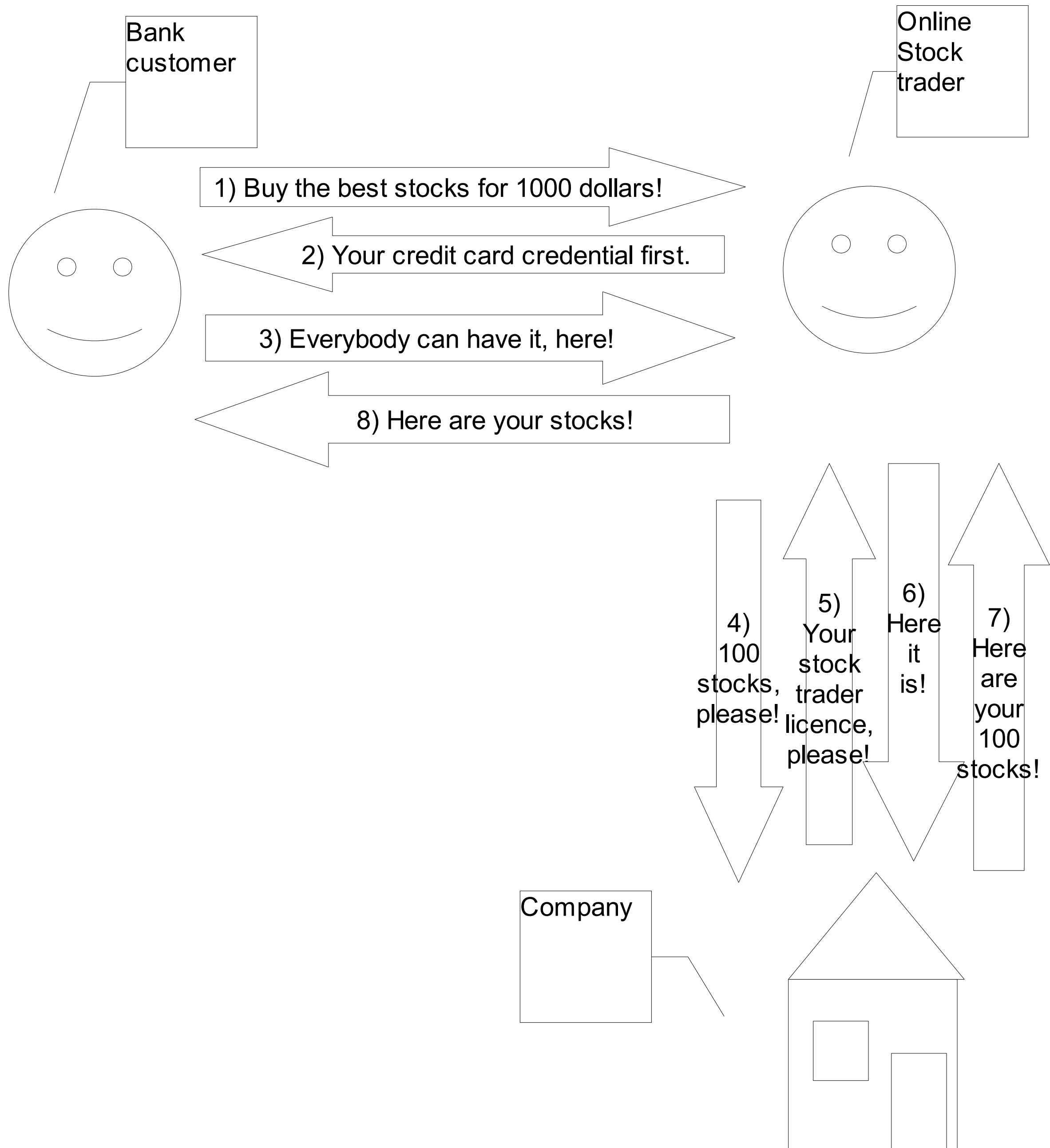
2.2.3 Examples



1. This small example for a trust negotiation is about a student who wants to apply for a Java exam online (the sensitive resource in this case) by his universitys examination office.
2. The office services sends a policy which requires a student and computer science credential to be shown first before applying to the Java exam (so the exam is protected).
3. The student does not want to show these credentials to anyone outside the university, so they are protected by a policy specifying this which is sent to the office.
4. The office recognizes that their university licence credential is not protected, so it can be given to the student without requirements.

5. His policy for the student and computer science credentials is satisfied now so he passes them back.
6. In the final step, all requirements for being applied to the exam are satisfied now by the student and he gets a confirmation that he applied successfully.

A second example shows an involvement of more parties in a trust negotiation.



1. A user wants to invest his money in stocks, so he asks an online stock trader.
2. The stock trader only helps him with buying stocks, if he discloses his credit card credential (for the purchase & billing).
3. The customer did not protect his credit card credential, so it is sent to the online stock trader.
4. The trader has found in his opinion a company which stocks are worth investing in and asks it if he can buy a certain amount.
5. The company requests a stock trader licence credential first before selling its stocks.
6. The trader did not protect his, so he gives it away with no requirements.
7. The purchased stocks are sent to the trader.
8. The trader forwards the stocks to the customer who requested them.

3 Challenge: How to protect my Web Resources

This chapter will introduce techniques that can be used to protect Resources on the Web, either statically (as a whole) or dynamically (specific content on the document level)

3.1 Motivating Scenario

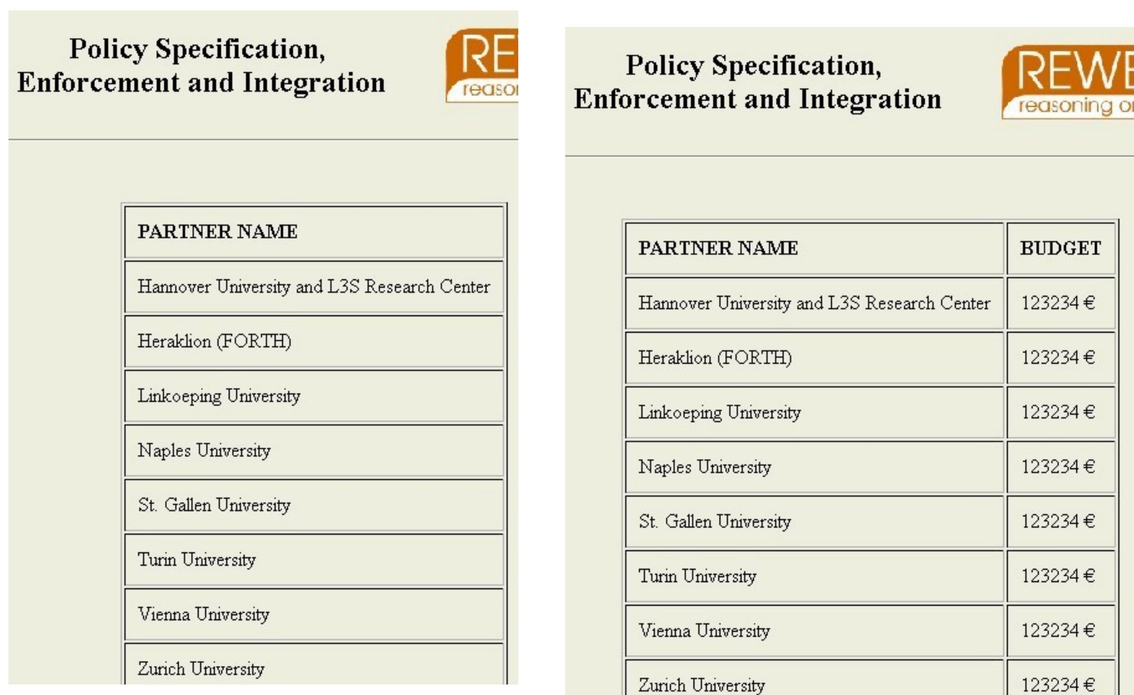
Adapting the trust negotiation approach for the traditional World Wide Web offers interesting possibilities. Imagine an example in which REWERSE [7] (a research network on „Reasoning on the Web“) wants to enhance its current website. It wants to offer content specially adapted to different kinds of user groups. Not everything should be freely available, some parts may only be accessed by REWERSE members, and going further, user who are also members of IEEE or ACM (computer societies, http://www.ieee.org and http://www.acm.org/) have their own sub parts in this REWERSE-only-section.

A first thought of realizing this concept with traditional registering/login-techniques was quickly rejected because of the disadvantages mentioned above. After a long discussion the REWERSE team decided to use trust negotiation to reach the intended aims. To distinguish which roles and privileges a user has (REWERSE-, IEEE-, ACM- or no member) and in order to make things easy for him, trust negotiation is automatically performed every time a protected part or resource of the website (REWERSE-, IEEE- or ACM-section) is request, policies on the server site guarantee that the user must present the appropriate credential in order to be granted access. Another team member proposed to not only apply trust negotiation for static documents, also dynamically generated pages may take advantage of this, as content of a single web page may be protected this way.

This allows multiple levels of content, for instance a non-member may see a small text when he accesses the page. A REWERSE-member may see additional

content (an enhanced version so to say) although he accessed exactly the same URL as the non-member. An IEEE- or ACS-member may see even more. Of course, also completely different content for each user group is possible on a dynamic web page. This eliminates the need to specify documents tailored to every member group and thus saves time and effort.

The following pictures illustrate this, the left one is a table of partners of REWERSE from the view of a non-member. In contrast to this, the right pictures is viewed by a REWERSE-member who can see an additional budget-column in the table. Both users have accessed the same dynamic web page at the same URL.



Policy Specification, Enforcement and Integration

PARTNER NAME
Hannover University and L3S Research Center
Heraklion (FORTH)
Linkoeping University
Naples University
St. Gallen University
Turin University
Vienna University
Zurich University

Policy Specification, Enforcement and Integration

PARTNER NAME	BUDGET
Hannover University and L3S Research Center	123234 €
Heraklion (FORTH)	123234 €
Linkoeping University	123234 €
Naples University	123234 €
St. Gallen University	123234 €
Turin University	123234 €
Vienna University	123234 €
Zurich University	123234 €

Simplicity is a fundamental prerequisite. The trust negotiation support and features described above should be available for the user with Zero-Install (no requirement to have a special software for the trust negotiation manually installed before) if he opens this REWERSE website with his browser. If the user is forced to install anything before, many will quickly loose interest and leave the site. Trust negotiations popularity will improve if the user recognize its benefits and realizes that there are no (relevant) disadvantages bound with it.

As another possibility to attract the users interest in trust negotiation and trust in its safety, REWERSE may provide them a free application which members can use to easily protect his resources (credentials, files etc.) with policies. This program should be simple, not too complicated and support the user (especially in writing and assigning policies). After having protected everything as intended, the next trust negotiation when visiting the REWERSE site takes these requirements of the user into account automatically without having the user to explicitly tell him. His member credentials can be protected this way for example. This way he controls what he wants to disclose and under what conditions. He gets the feeling that although trust negotiation is performed automatically, he still has control and nothing is done against him.

The next part will focus on the main challenges of trust negotiation in the World Wide Web in more detail. The above example already introduced them.

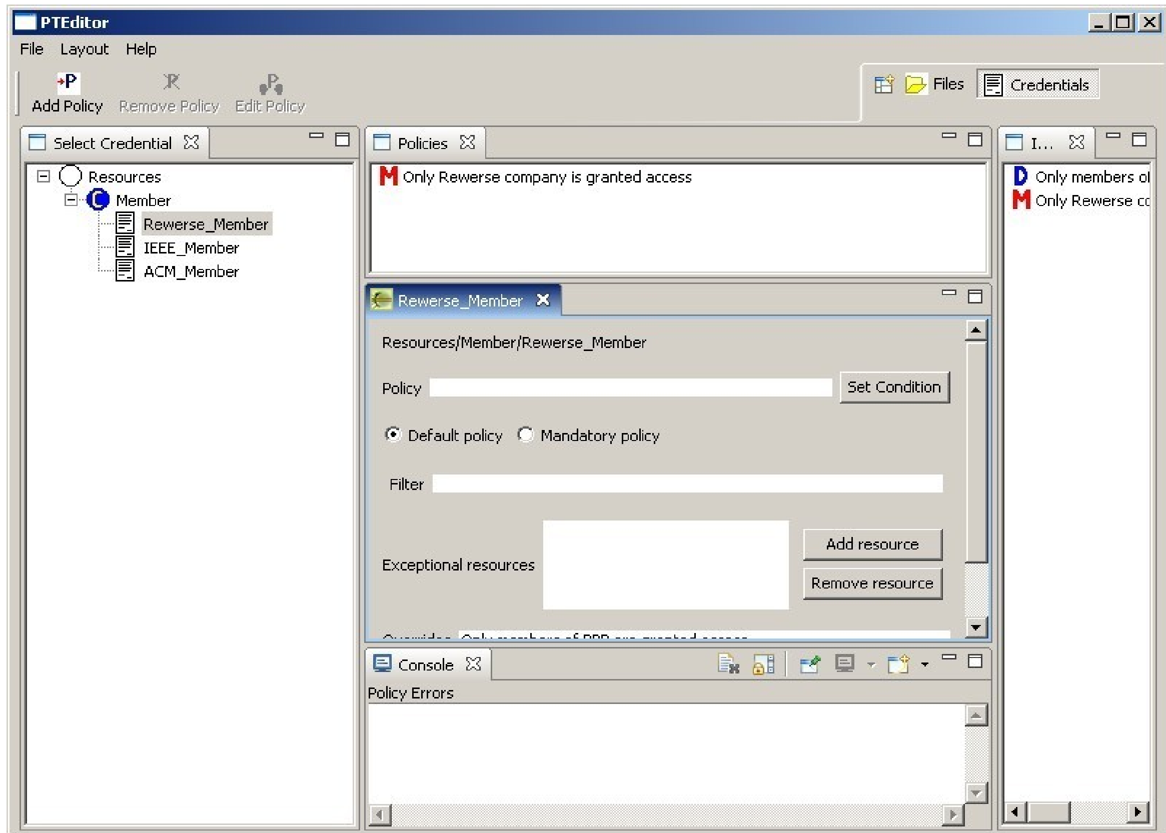
3.2 Challenges

The aim of this thesis was making a scenario like the example above possible. Three main tasks have been identified which will be described in more detail in the following chapters (4 to 6).

3.2.1 Policy Management Tool

Making trust negotiation easy was already judged as a key prerequisite. Allowing either clients and servers to protect their sensitive resources (credentials, files etc.) easily (without required knowledge) will decrease their scepticism (concerning security issues) in trust negotiation and giving them control despite the automatic nature of trust negotiation. This will help trust negotiation to become more and more widespread.

Special software tools may guide the user in writing policies and assigning them to resources. One of the tasks in this thesis consisted of designing and implementing such an policy management tool. It provides the native look and feel for the users operating system and an easy-to-use GUI with drag and drop support. Chapter 4 will cover the tool in more detail. An screenshot can be seen below.



3.2.2 Policy-Based Dynamic Content Generation

This mechanism allows the protection of content. A user must satisfy the assigned policy by for instance disclosing the matching credentials. This may be used to diminish the number of pages stored at the server, as many user groups can now use the same page instead of individually tailored versions. Multiple levels of content or completely different ones can be realized. The content is generated according to policies.

Imagine dormitories (house with apartments just for students) in a town for which a web page was created about the internet accounts in these buildings. Students, administrators and the student union exists as user groups which may all access the same page but the content and actions allowed are different. A student can just check if his internet account is blocked or not (because of too much download,

spamming or virus infection), administrators and the student union can see all internet accounts in the buildings, but only the administrator can create, edit or delete them.

A requirement for protecting parts is that the web page itself is dynamic, if a user requests it, the servers generates one (instead of just using static content) and sends it back. For each protected part, a trust negotiation with the client will be triggered during this generation process. Depending on their success or failure, the different parts are integrated or left out in the finally generated page. So the web designer responsible for creating a web site has to make a trade-off when considering using this dynamic protecting mechanism. The benefit of reducing the site of pages stored on the server comes with the price of higher traffic, as it may take longer to access a page with protected parts as for each one a trust negotiation must be performed.

This task is described in more detail in chapter 5.

3.2.3 Negotiating on the Web

Besides the trust negotiation support in dynamic web pages (to protect content) which was covered in the previous section, the server architecture developed for this thesis allows also the protection of whole pages or documents in a web site. So instead of having to register and login to access a protected resource like in the traditional approach, a client must satisfy the policy protecting it (e.g., by disclosing the right credential).

Premium services can be realized this way (imagine a web page of a newspaper in which very few particular articles are open for everyone. Only subscribers who get the newspaper regularly by post may access a full pdf version on the web page. When they applied for subscription, a special member credential was given to them (in a secure channel) which they must show if they want to get the newspapers pdf file. Another possibility of protecting resources would be that the newspaper allows subscription only for special categories (sports, politics etc.) and access the relevant document by disclosing the category-credential.

While the server architecture developed in this thesis already supports trust negotiation, normal browsers and the http protocol do not, so it has to be made available to the user when he accesses the web page. As a trust negotiation agent has to act on his behalf, a piece of software must run in the background. Requiring the user to manually install such a software is not very comfortable and may lead users to loose interest and not browse this site any more. So it would be better if the clients browser installs the software hidden from the user (he would not recognize it) when accessing the web page for the first time. If the software itself is also not visible on the screen, the user would not see any difference to browsing pages without trust negotiation. One thing the software must do is somehow interfere the browsing behaviour of the user, so that the trust negotiation can effectively take place if the user clicks a link (as the destination may be protected). Section 6 will cover this theme.

This negotiation software on the client side, as well as the equivalent on the server need to know about the sensitive resources and the applied policies of the particular side, so they use the output file that can be generated with the policy editor previously described.

4 Policy Management

As already mentioned, writing policies and assigning them to sensitive resources like credentials or files is a main requirement in trust negotiation. In order to make the latter more popular, the L3S Research Center [3] investigates how this task can be made as easy as possible for the user, so that also non-experts and beginners may quickly write and assign their policies without spending hours on learning the policy-syntax or manually assigning their policies to resources in complex text files. Instead, special tools or wizards may guide and assist the user. One task of this thesis was designing and implementing such a tool on which this chapter focusses. It may be used to assign policies to arbitrary files or credentials.

4.1 Policies

Policies can be seen in case of trust negotiation as special rules that specify what another party will have to do if it wants to access the resource to which this policy was assigned. The L3S [3] developed the policy language PeerTrust [1] which is used for the testing of the implementation of this thesis, but the exact syntax will not be covered here, as it is out of the scope for this thesis (an editor for writing policies was not part of this thesis). Furthermore the approach is language independent.

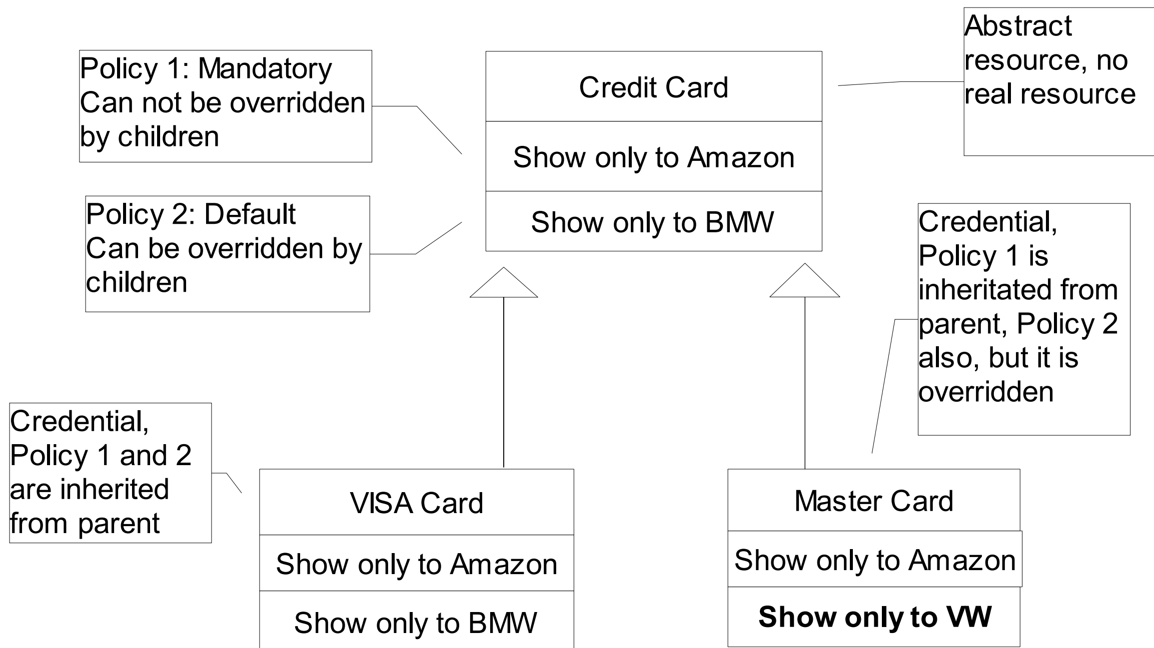
4.1.1 Assigning policies to resources

Assigning policies to resources is not trivial in all cases. If a large amount of resources should be protected, allowing only the assignment of policies to each single resource is too restrictive and complicated (imagine if someone wants to assign a special policy to 100 of his 1000 resources, or delete it from 50 later). The system allows building and using hierarchies of resources. Like attributes and methods of classes in object oriented languages (Java, C++ etc.) policies may be

inherited (no multiple inheritance is allowed). This simplifies policy assignment, as when assigning a policy to a resource, all its children are also affected. In order to build a hierarchy, special abstract resource classes serve as the main building block (as a resource and classes may be assigned to it as children). To enhance possibilities further, the tool allows to specify if inherited policies of a resource can be overridden by its children or not (like the absence or presence of the final-keyword in a method declaration Java). So called default policies allows it, while mandatory policies do not [5].

Besides supporting inherited (mandatory and default) credentials and the possibility to override, the tool offers filters and exceptions to make assigning policies even easier. Filters restrict to which children of a resource a policy applies. If someone has a folder in his file system and wants to protect all pdf files in it with a policy, he can not do it as there are also other files (like txt or jpg) and the so the pdfs can not inherit the policy from the folder. Filters may help here, as one may assign the policy to the folder and restrict the children which inherit it by specifying a filter which points to all pdf files (for example „/*.pdf“), other files would not be affected then. Specifying exceptions for a policy is helpful, if someone already uses a filter (not necessary), but wants special affected children to be excluded. In the example with the folder, a user may want to protect all his pdf-files, but not the jokes.pdf, as he thinks it is funny and and not harmful at all, so everyone may access it without any restrictions.

4.1.1.1 Example



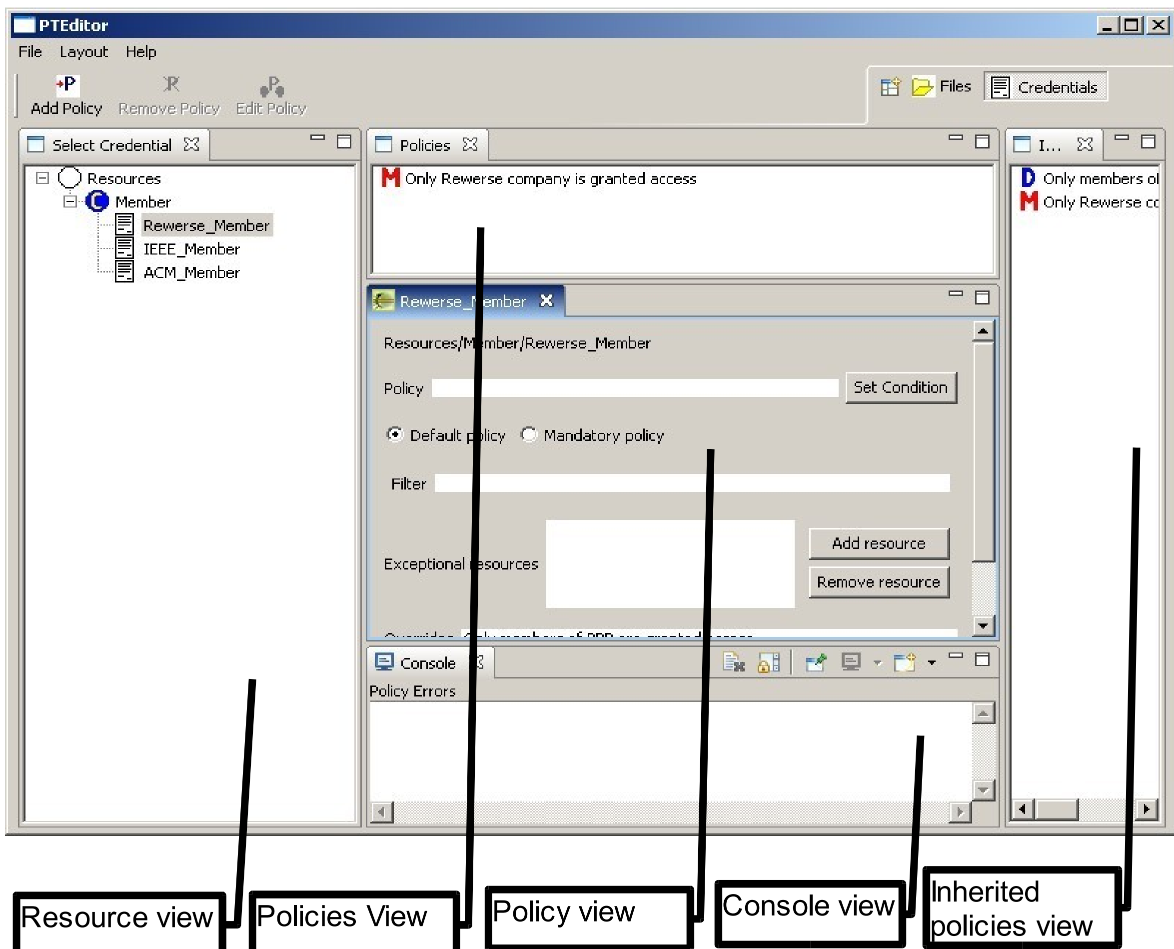
The above example shows a hierarchy. A user wants to protect his VISA and Master Card credentials with policies. To save work, instead of assigning them directly to the credentials, he creates a resource class named „Credit Card", because he recognizes that the policies for his two credentials are very similar. As the credentials inherit the policies from the parent class, he assigns them there. Policy 1 is a mandatory policy, so it can not be overridden by the resources children. On the other hand, the user does not want to use his Master Card in a transaction with BMW, so he overrides policy 2 (this is allowed because it is default) there. No policies must be assigned for his VISA Card credential as it inherits everything from the „Credit Card" resource class. Instead of overriding a default policy, it may also be ignored.

4.2 *The Policy Management Tool*

The policy writing and assignment tool will be covered now. After introducing it, technical and implementation details follow.

4.2.1 Description

The following screenshot shows the tool in action. On the upper right the user has the choice to switch between two possible perspectives, for files and credentials. That is because it is possible to assign policies to these two kinds of resources (other types are not supported yet). So if someone wants to protect his credentials, he selects the credential-, otherwise the file-perspective. These two modi will be covered later in more detail.

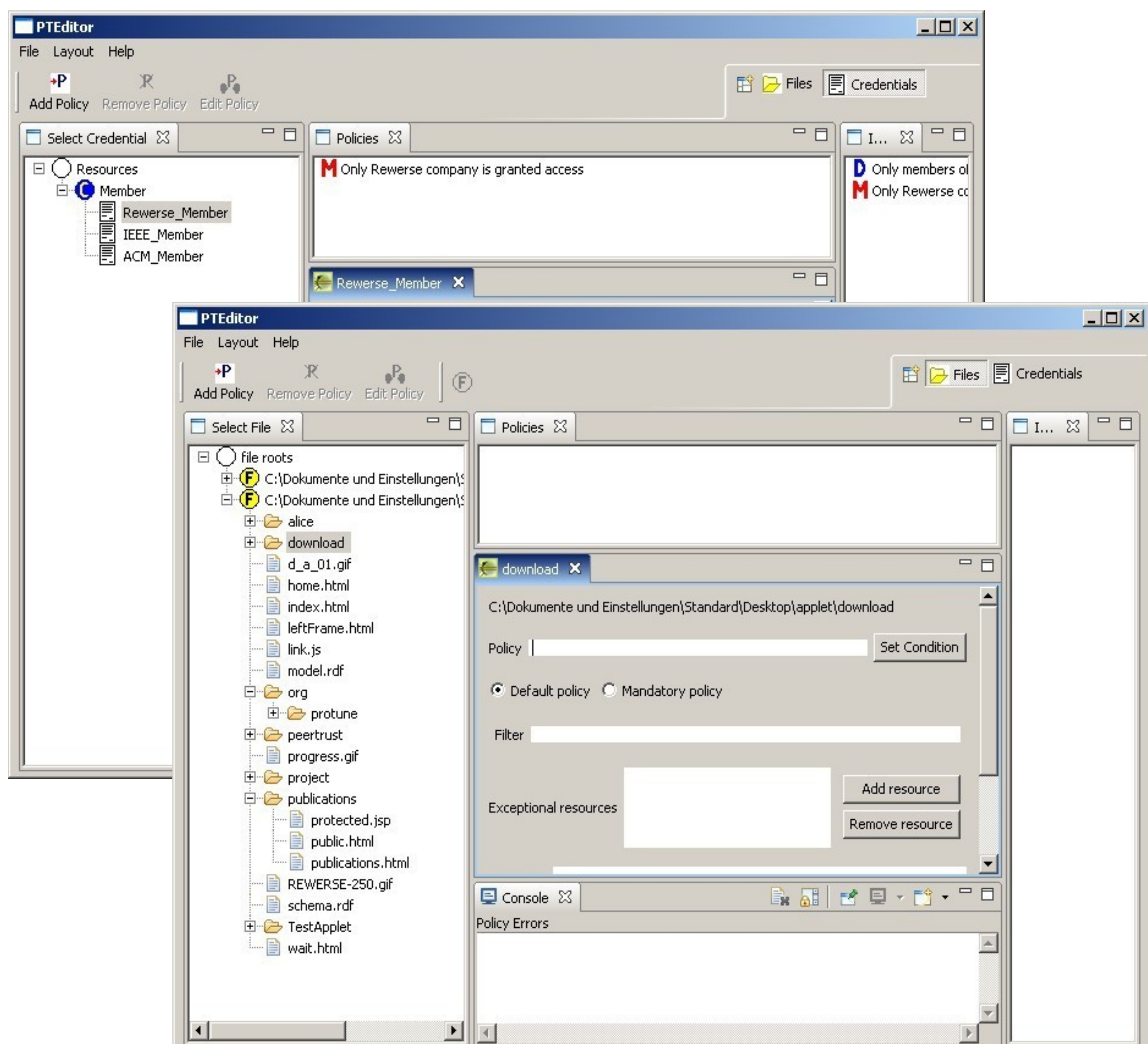


The main window is divided into five views which consist of

1. The resource view
2. The policies view
3. The policy view
4. The console view
5. The inherited policies view

4.2.1.1 The Resource View

The resource view shows either all credentials or files (depending on the currently selected perspective) to which policies can be assigned right now.



The upper screenshot shows the resource view in the credential-, the lower in the file-perspective.

In order to avoid confusion and to have only the relevant resources at a glance, no credentials and files from the user can be seen after the program was started. Instead one can add resources to the view by either importing credentials (e.g., from a keystore) or add file directories (if one wants to protect files of his web server, one may only want to see the directory they are in instead of all files of the hard drive).

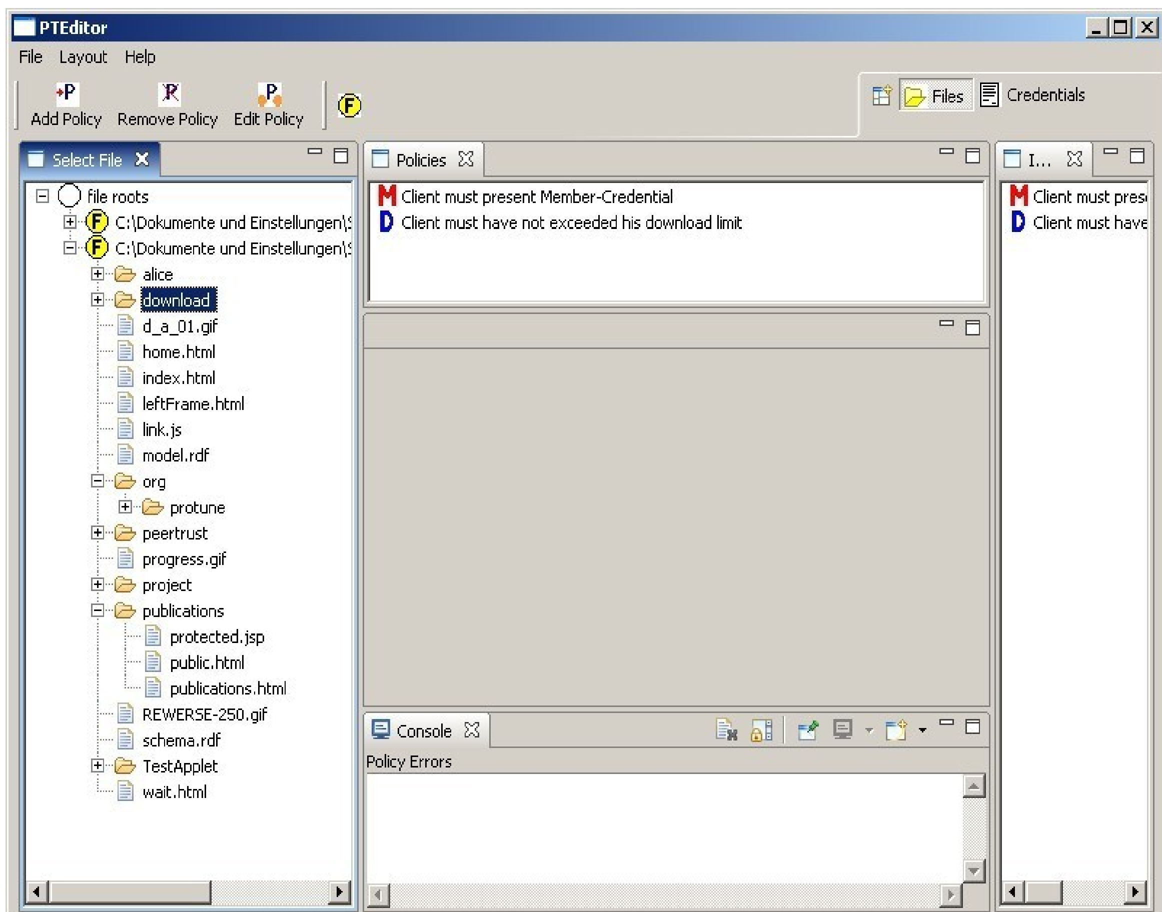
The resources view of the credential- allows further user interaction than in the file-perspective. Building of credential hierarchies (see section 4.1.1) allows the creation of resource classes items (the „Member“-item in the above screenshot is one) and assigning children to their parents (via drag and drop). This is not possible for the file-perspective, as the hierarchy for the files is already given by the users hard drive.

If a resource should be protected, it has to be selected in the view. The policies which are already applied for this resource now become visible in the „Policy“- and „Inherited Policies“-views where they may be erased or edited. Of course, also a new policy can be added to the resource. That is where the policy editor comes into play.

4.2.1.2 The Policies View

This view shows all policies that are directly assigned to the resource selected in the resource view (the policies inherited from a parent resource would not appear here, this is the task of the „Inherited policies view“). The symbol „D“ or „M“ identifies them as being default or mandatory. The listed policies here may be edited or deleted, or new ones can be added to the resource. In the first and last action, the policy editor is involved.

The following screenshot shows the policies view. The user wants to protect files on his web server and has assigned two policies for the selected download-folder (and so also for all its children) in the resource view. The first one is mandatory and specifies that a client has to disclose a special member-credential in order to access the folder and its content. The second policy is default and requires the client not to have exceeded his download limit before grant him access.



4.2.1.3 The Policy Editor

The policy editors purpose is assigning a policy. It is located in the center of the main window, but in contrast to the other views, it appears only when needed (if a user has selected to add or edit a policy), otherwise it can not be seen (like in the screenshot above). After having changed or added a policy with the editor, the

„Policies“ and „Inherited Policies“-Views are immediately updated to reflect the change.

The editor itself allows specifying

- The policy itself
- Shall the policy be mandatory or default (refer to section 4.1.1) [5]
- A filter (see section 4.1.1)
- Exceptional resources (refer to section 4.1.1)
- Policies of parents that this credential overrides (see section 4.1.1)

4.2.1.4 The Console View

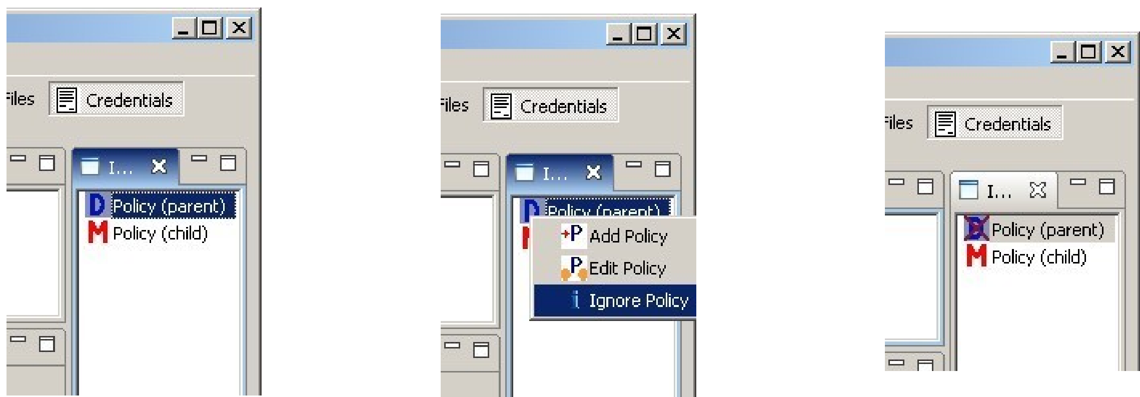
The task of the console located under the policy editor is to inform the user if anything went wrong and offers solutions to fix it.

Although currently being not of great help, the use of the console may be enhanced in the future to display more errors, warnings or hints.

4.2.1.5 The Inherited Policies View

The inherited policies view is basically identically to the policies view. It has also the same representation (default- or mandatory-policy-symbol followed by the policy as text). The main difference is that the policies view only displays the policies that are directly assigned to the selected resource. The inherited policies view however displays also the policies inherited from the resources parents. So a user can directly see which policies apply for the resource he selects without having to browse the parents.

Section 4.1.1 introduced the possibility to specify exceptions for the children of a resource in a policy. If the currently selected resource is an exception of one of the policies inherited from the parents, the affected policies are marked with an striked through symbol. So the user has the ability so see, if this resource as an exception of a policy from a parent. He may also change this by himself. Instead of having to edit the parent policy to specify the current resource as an exception, he may directly mark the parent policy in the inherited policies view as having this resource as an exception or not with an entry in the views context menu. The following screenshot illustrates this.



On the outer left screenshot, two policies appear in the inherited policies view, the first is inherited from the parent, the second is directly assigned to the resource selected. In the second screenshot the user wants to specify this resource as being an exception of the policy from the parent (so the policy should not apply for its child) and uses an extra in the context menu to do this (if the policy would be mandatory, the entry will be disabled). In the last screenshot, the symbol of the parent symbol has changed, it appears as being striked through and marks that this policy is inherited, but does not apply, because the selected resource is an exception of it.

4.2.1.6 RDF Export and Import

The main purpose of this tool was to assign policies to the resources that should

be protected in trust negotiation, so the user should be able to save his work. The relevant information will then be stored in an RDF file (Resource Description Framework) [8] which can then be passed to his trust negotiation software. Of course this RDF file can be loaded into the tool again to modify it.

The following code shows an extract of the schema for this RDF files.

```
<?xml version="1.0" encoding="utf-8"?>

<rdf:RDF xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
             xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
             xmlns:owl="http://www.w3.org/2002/07/owl#"
             xml:base="http://sourceforge.net/projects/peertrust/#">

      <owl:Ontology rdf:about="">
            <rdfs:label>Resource Protection Ontology</rdfs:label>
      </owl:Ontology>

      <rdfs:Class rdf:ID="Root">
            <rdfs:label>Starting point</rdfs:label>
      </rdfs:Class>

      <rdf:Property rdf:ID="startingPoint">
            <rdfs:domain rdf:resource="#Root"/>
            <rdfs:range rdf:resource="#ProtectedResource"/>
      </rdf:Property>

      <rdfs:Class rdf:ID="ProtectedResource">
            <rdfs:label>Protected Resource</rdfs:label>
      </rdfs:Class>

      <rdf:Property rdf:ID="type">
            <rdfs:domain rdf:resource="#ProtectedResource"/>
            <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Literal"/>
      </rdf:Property>

      <rdf:Property rdf:ID="policy">
            <rdfs:domain rdf:resource="#ProtectedResource"/>
            <rdfs:range rdf:resource="#Policy"/>
      </rdf:Property>

      <rdf:Property rdf:ID="subitem">
            <rdfs:domain rdf:resource="#ProtectedResource"/>
            <rdfs:range rdf:resource="#ProtectedResource"/>
      </rdf:Property>

      <rdf:Property rdf:ID="parent">
            <rdfs:domain rdf:resource="#ProtectedResource"/>
            <rdfs:range rdf:resource="#ProtectedResource"/>
      </rdf:Property>
```

```
	<rdf:Property rdf:ID="domain">
		<rdfs:domain rdf:resource="#ProtectedResource"/>
		<rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Literal"/>
	</rdf:Property>

	<rdfs:Class rdf:ID="Policy">
		<rdfs:label>Policy</rdfs:label>
	</rdfs:Class>

	<rdf:Property rdf:ID="filter">
		<rdfs:domain rdf:resource="#Policy"/>
		<rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Literal"/>
	</rdf:Property>

	<rdf:Property rdf:ID="exception">
		<rdfs:domain rdf:resource="#Policy"/>
		<rdfs:range rdf:resource="#ProtectedResource"/>
	</rdf:Property>

	<rdf:Property rdf:ID="default">
		<rdfs:domain rdf:resource="#Policy"/>
		<rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Literal"/>
	</rdf:Property>

	<rdf:Property rdf:ID="override">
		<rdfs:domain rdf:resource="#Policy"/>
		<rdfs:range rdf:resource="#Policy"/>
	</rdf:Property>
</rdf:RDF>
```

After this short explanation of how the tool looks like and works, the following part will delve into a more technical level and will explain the technologies used and relevant implementation details (main algorithms, etc.).

4.3 Implementation

In order to support a wide range of different operating systems and ease its distribution, the Java programming language [9] was chosen, as its generated byte code can be processed by virtual machines which are available for nearly every platform or operating system. In order to provide native look and feel and allow extensibility, the tool uses the Rich Client Platform (RCP) [10,16] from Eclipse [11].

4.3.1 Rich Client Platform

Although the GUI-APIs Swing and AWT [9] that Java provides in the Java Development Kit (JDK) are easy to use, the controls often do not look native to the operating system they run on. The user often sees at a glance if an application is written in AWT or Swing, because the look and feel is different from native applications on his computer. That is because the APIs do not use the native controls of the system, instead they emulate them.

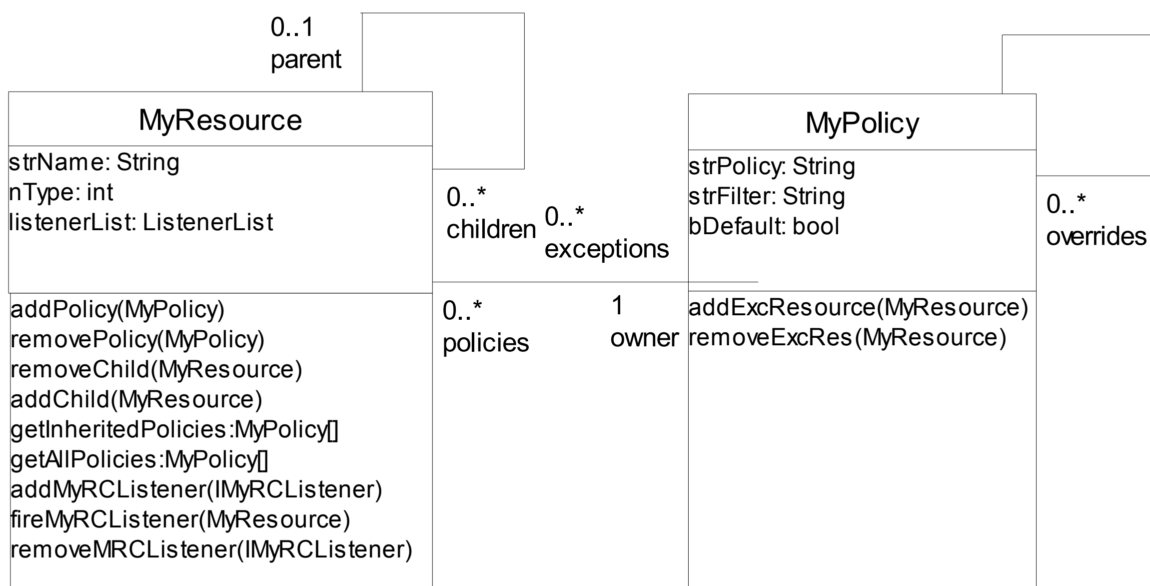
Since its release, the famous development environment Eclipse from IBM had an great impact on the Java scene. The company quickly recognized the disadvantages of AWT and Swing and developed a custom API for GUIs, the Standard Widget Toolkit (SWT) [12]. This library attempts to use the native controls of the OS directly and so the applications developed with it have a more native touch than the previous APIs do. This does not come without a price, however, as the SWT library is not platform independent any more, it is adapted and must be installed for different operating systems.

The Rich Client Platform (RCP) [10,16] was also developed by the Eclipse team and uses SWT. This framework allows the use, composition and creation of Plugins. All components from Eclipse (for example text editors, wizards, help system etc.) can be used in custom applications this way and allow the developer

to spend more time on the features and logic of his software. Complex and good-looking software can be realized via RCP without having to reinvent the wheel and updating is easy. RCP combines the advantages of thin and fat clients. Plugins may cooperate or can be configured over so-called extensions (-points) which use XML. So if a developer wants to add a new entry to the applications menu bar for example, he does not need to write Java code for it, instead he may specify it declarative with XML. RCP offers many Plugins for the GUI (views, editors, menu bar, tool bar, consoles, perspectives etc.) that may be used and configured appropriately.

4.3.2 Model Implementation

The tool is about assigning policies to resources so these two parts can be identified as the key classes of the model. They are completely independent from the GUI. The following UML class diagram gives a short overview of the important attributes and methods (the getter- and setter-methods of the attributes are ignored here). The listener that can be registered to MyResource is also not displayed here.



An object of the MyResource class may have one parent resource or not (in this

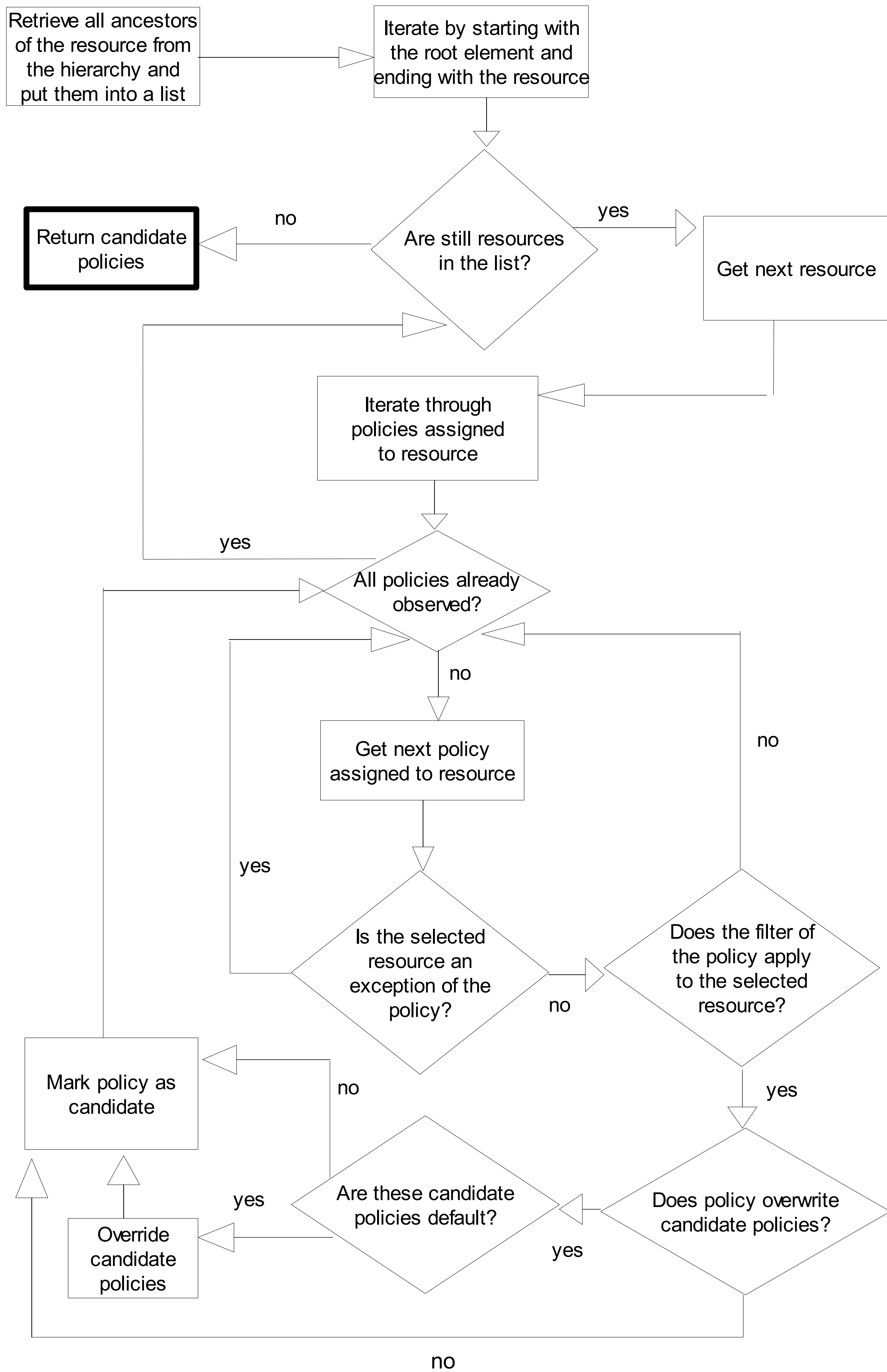
case it would be the root resource). Additionally it may have an arbitrary number of children and policies. Furthermore it has a name and a type (which specifies if the resource is a file, a directory, a resource class, a credential, etc.). Besides the usual getters and setters for the attributes, it allows the addition and removal of new child resources or policies and supports the registration, activation and removal of a special listener. This was done so that the GUI can itself register as a listener and may so be notified of changes in the hierarchy, on which it may react by redrawing, etc.

The MyPolicy class on the other hand has an owner (a resource), a policy and the type (mandatory or default policy). In order to satisfy the concepts discussed in section 4.1.1 it may also contain exceptional resources and policies that should be overridden. Methods allow to add and remove exceptional resource besides the usual setters and getters.

4.3.2.1 Retrieving Policies for a Resource

While the methods of both classes are more or less simple, the getAllPolicies()-method of the MyResource class is the most important one, as it used by the trust negotiation agents later to find out which policies apply for a resource. Therefore it is worth being mentioned here.

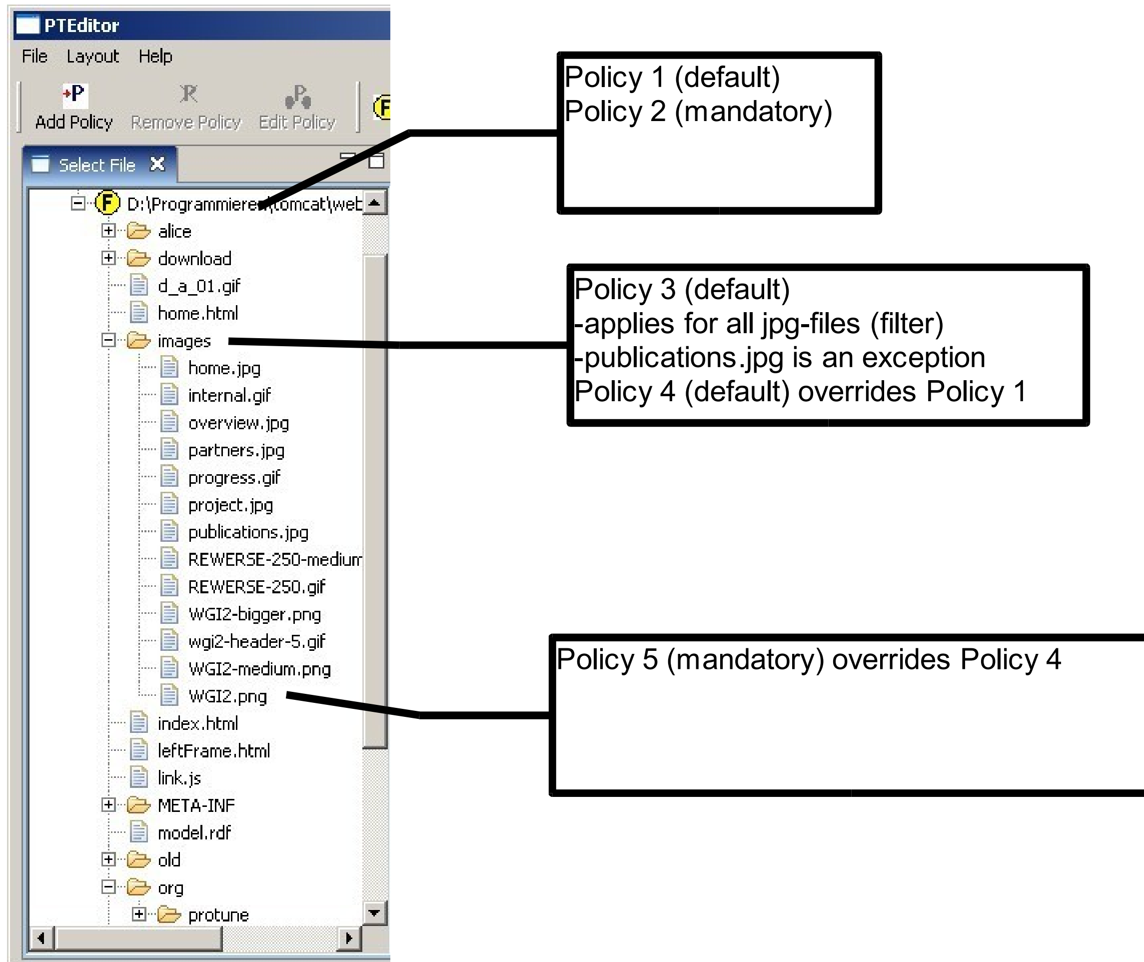
The problem is that it is not sufficient to only take the policies directly assigned to the resource into consideration, as all resources up the hierarchy (the parents) have to be checked as well. This is not trivial, because mandatory and default policies, exceptions, filters and the overriding of policies also come into play and make things difficult. This section will introduce an algorithm which can solve the task, enriched with examples.


Retrieve all ancestors of the resource from the hierarchy and put them into a list
Iterate by starting with the root element and ending with the resource
Return candidate policies
no
Are still resources in the list?
yes
Get next resource
Iterate through policies assigned to resource
yes
All policies already observed?
no
Get next policy assigned to resource
no
yes
Is the selected resource an exception of the policy?
no
Does the filter of the policy apply to the selected resource?
Mark policy as candidate
no
yes
yes
Are these candidate policies default?
Does policy overwrite candidate policies?
Override candidate policies
yes
no

The above diagram shows how the algorithm works. The selected resource is the input (parameter) and the output is the list of policies which apply for it. So the algorithm needs to check the resource and all of his ancestors (the parent, the parent of the parent, etc.), in top-down manner (the root first and the input resource last). If no resources have to be checked any more, the algorithm returns the policies he has collected so far, otherwise all assigned policies from the current resource will be observed. If the current policy has specified an exceptional resource which fits to the input parameter, the policy does not apply. This is also the case if the policy has a filter which does not include the input resource. If otherwise the policy passes this two tests, it is marked (but only as a candidate). If it overrides any policies which are also marked as candidates, those will loose this status if they are default policies (as mandatory ones can not be overridden).

4.3.2.2 Example

The following example shows a file hierarchy in which some resources are protected by policies, the first entry is protected by Policy 1 and Policy 2, the folder „images“ by Policy 3 (filters all jpg-files, with „publications.jpg“ as an exception) and Policy 4 (which overrides Policy 1) and the file „WGI.png“ by Policy 5 (which overrides Policy 4).



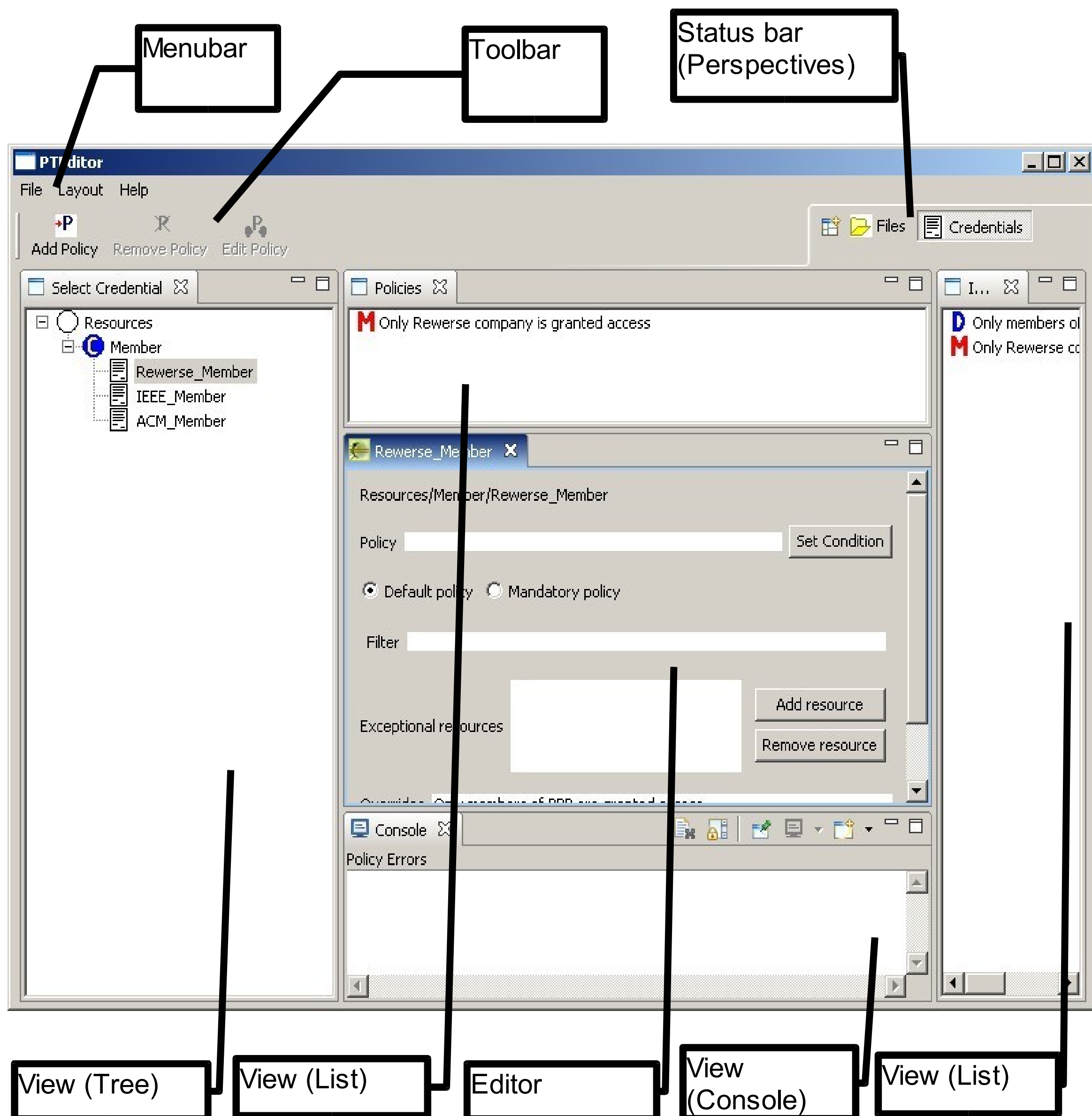
The following table shows how the algorithm behaves with two different input resources.

Input resource	Operation	Candidate policies
alice (folder)	Get policies assigned for parent of alice (D:\Programmieren\tomcat...) -> Policy 1 and Policy 2	-
	alice is no exception for Policy 1	-
	Policy 1 has no filter	-

Input resource	Operation	Candidate policies
	Policy 1 does not overwrite policies	-
	Policy 1 is marked as candidate	Policy 1
	alice is no exception for Policy 2	Policy 1
	Policy 2 has no filter	Policy 1
	Policy 2 does not overwrite policies	Policy 1
	Policy 2 is marked as candidate	Policy 1, Policy 2
	Get policies assigned for alice ->None	Policy 1, Policy 2
	No other resources to check -> Algorithm finished	**Policy 1, Policy 2**
WGI2.png (file)	Get policies assigned for root (D:\Programmieren\tomcat...) -> Policy 1 and Policy 2	-
	WGI2.png is no exception for Policy 1	-
	Policy 1 has no filter	-
	Policy 1 does not overwrite policies	-
	Policy 1 is marked as candidate	Policy 1
	WGI2.png is no exception for Policy 2	Policy 1
	Policy 2 has no filter	Policy 1
	Policy 2 does not overwrite policies	Policy 1
	Policy 2 is marked as candidate	Policy 1, Policy 2
	Get policies assigned for images (parent of WGI2.png) -> Policy 3 and Policy 4	Policy 1, Policy 2
	WGI2.png is no exception for Policy 3	Policy 1, Policy 2
	Filter of Policy 3 does not apply for WGI2.png (no jpg-file) -> Ignore Policy 3	Policy 1, Policy 2
	WGI2.png is no exception for Policy 4	Policy 1, Policy 2
	Policy 4 has no filter	Policy 1, Policy 2
	Policy 4 overrides Policy 1 -> Policy 1 is default -> eliminate	Policy 2
	Policy 4 is marked as candidate	Policy 2, Policy 4
	Get policies assigned for WGI2.png -> Policy 5	Policy 2, Policy 4
	WGI2.png is no exception for Policy 5	Policy 2, Policy 4
	Policy 5 has no filter	Policy 2, Policy 4
	Policy 5 overrides Policy 4 -> Policy 4 is default -> eliminate	Policy 2
	Policy 5 is marked as candidate	Policy 2, Policy 5
	No other resources to check -> Algorithm finished	**Policy 2, Policy 5**

4.3.3 GUI Implementation

As already mentioned into the introduction of Rich Client Platform [10,16] (section 4.3.1), existing plugins may be used by a developer to compose the application he has in mind. This may save work tremendously, as the developer can skip time consuming and error prone tasks (in favor on concentrating on the main features or the business logic) this way by using plugins which contain the desired functionality. Graphical User Interfaces (GUI) are an example in which plugins may help to create them quickly, easily and good looking.



The policy assignment tool makes use of such GUI Plugins and components that Eclipse offers. The screenshot above shows a workbench window which consists

of a menu bar, a tool bar, a status bar and the main area which may consist of views, editors, etc.

Perspectives define the layout of the workbench windows main area. Beside offering distinct options (accessible via the menu bar, tool bar, context menu etc.), they may specify the views to be shown and their position. In the policy assignment tool, two different perspectives exist. The first one is the file perspective in which the user may assign policies to files or folders, the second one focuses on credentials instead. These two perspectives are similar, as they share the Policy View (above editor), the Inherited Policies View (right to edtor) and the console (below editor). The main difference are the different options they offer (this topic will be treated later) and the left view in which the resources are shown (Resource View).

Each perspective uses a special tailored view, as the content of both have not much in common. The credential perspective needs a view which shows credentials and abstract resource classes, whereas the file perspective displays files and folders. Both views contain a tree, but the content and the symbols displayed is completely different. Additionally, the view with the credentials allows the user to build a credential hierarchy via drag and drop. This is not possible in the view with the files, as the file hierarchy is already given by the users file system.

Both views use the model class MyResource (section 4.3.2) for the hierarchy of the tree. This is done by RCP wrappers which also allow to specify an icon and text for each individual resource (as the MyResource class contains no GUI specific code at all). The view defines a listener which is registered to objects of the model, so the tree can be notified if a resource is deleted, added or changed and react accordingly (by repainting for instance to reflect the changes). The drag and drop support in RCP is very similar to its counterpart in AWT in Java, as an object has to be transformed to bytes when the drag starts and reconstructed from it after the user dropped the item, methods which allow exporting to and importing

from a byte-array have been added to the MyResource (model) class.

The Policy and Inherited Policy View are simpler than the ones from the previous section, but share more or less the same approach. They consist of a list instead of a tree and as they display only policies, they use the MyPolicy class (section 4.3.2) which is again integrated into the list via RCP wrappers which provide an optical representation of these objects in the listbox (text and icons).

The views itself are independent, they hold no references to other views nor does a global registry exist in which they may be accessed. In order to enable communication between them anyway (for instance if a user clicks on a resource in the credential view, the policies assigned to it should appear immediately in the Policies and Inherited Policies Views), an existing RCP mechanism was used.

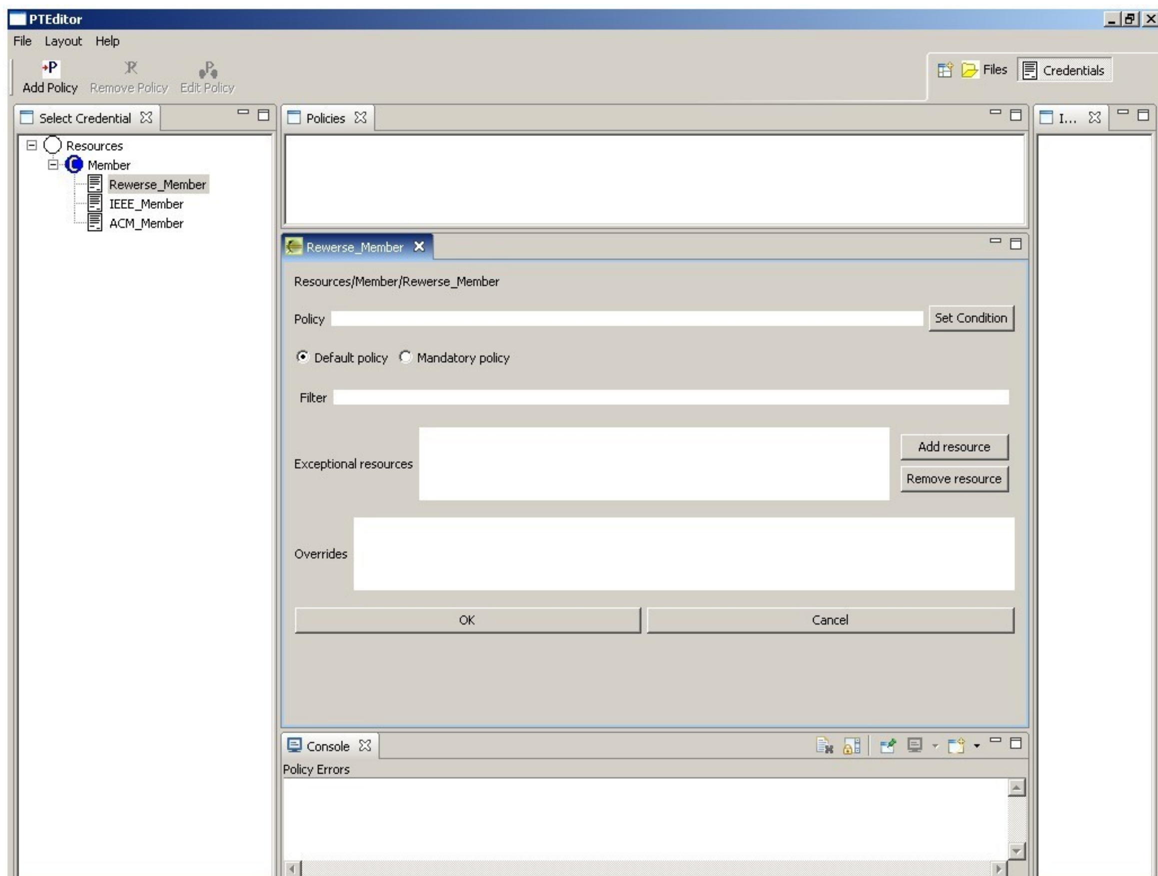
Each view that supports it may register himself as a so-called selection provider. If a user selects a new item in the view (a tree, list etc.) this selection event will be distributed to the workbench window. Every class can be notified of these events via special listeners (the workbench window serves as a kind of transmission channel), so communication between views can be easily realized using this mechanism.

Returning to the above example, the credential view may register himself as a selection provider and the Policies and Inherited Policies Views as selection listeners. The event contains the selected object, so the Policies View may easily extract the policies out of the received resource object. Each view of the policy assignment tool registers himself as n selection provider and listener. Even though not used by everyone, this allows extensibility for future use.

The editor is a special view of the workbench window that allows the user to modify data. In the assignment tool, the MyPolicy (model) objects that are bound to the selected resource may be modified. If a user clicks on the OK-Button of the

editor, the model is updated and the views are again notified by the selection provider and listener mechanism just introduced to reflect the changes immediately.

The editor offers to assign the policy, specify if it is mandatory or default, choose exceptional resources and policies that should be overridden. A screenshot can be seen below. Important is that the editor is only opened when needed and more than one editor can be opened at the same time (tabs exist to select the appropriate instance then).



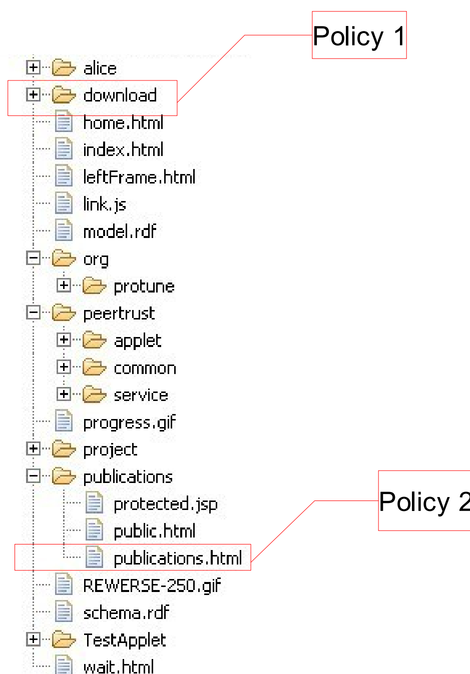
4.3.4 Logic Implementation

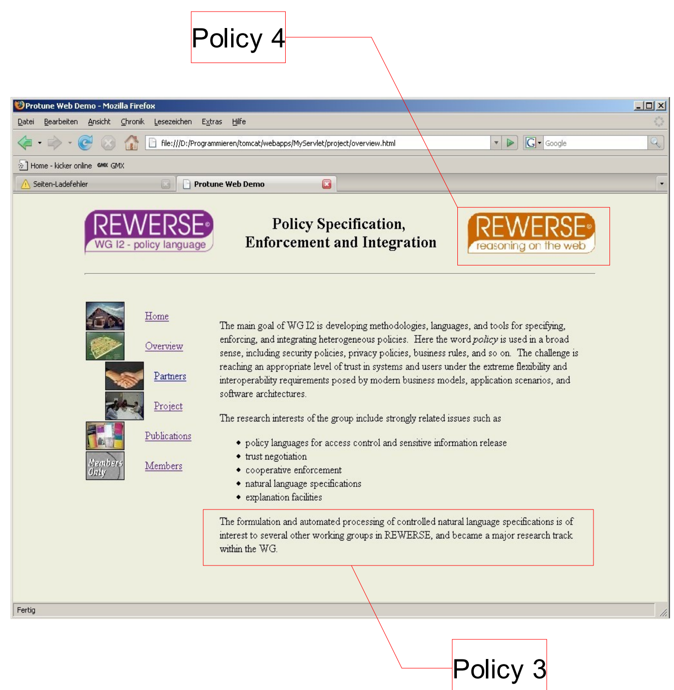
To distinct GUI, model and logic, so called Actions can be used in RCP. An action is a sort of command or feature that the user may invoke in the application. Examples are opening a file, creating or deleting a new policy or adding credentials or files to the resource views. Each command may be wrapped in an action class which contains the implementation of the action that should be performed in case of activation of the command. Each command can be given an icon and name. Furthermore, it may be specified where the command should be placed (in the menu bar, the tool bar, a context menu etc.). All this information can be directly specified in code or via extension points in XML. This way, new plugins can enhance an existing application, as the new actions will be added to the workbench accordingly. Many actions have been integrated into the policy assignment tool (for instance the possibility to add new files, credentials or resource classes, to add, edit, or delete a policy, to import from or export to an RDF file, etc.). As actions may also register themself as selection listeners, it is possible to enable them only if for instance a special resource is selected. For example the action for adding a new policy makes no sense if no resource is selected.

5 Policy-Driven Protection & Dynamic Content

Another task of this thesis consisted of providing an infrastructure which allows people who run a web server to protect their web documents (HTML, PDF, JSP, images etc.) with policies. This way, the traditional registering/login-approach can be replaced. If users want to access sensitive resources (for instance a page only for premium members), they will have to satisfy the assigned policies (by disclosing the matching credentials, etc.).

Protection of web resources can be performed at two different levels. The first one allows to protect documents as a whole (an HTML-page, a folder, etc.). This is closely related to the policy assignment tool described in the previous chapter. The second one delves into the level of single web pages, in which a developer is able to protect certain content with policies. This way, he may reduce the number of pages stored on his server (instead of providing different page versions for different user groups, a unified one is enough) and may enrich his web content.





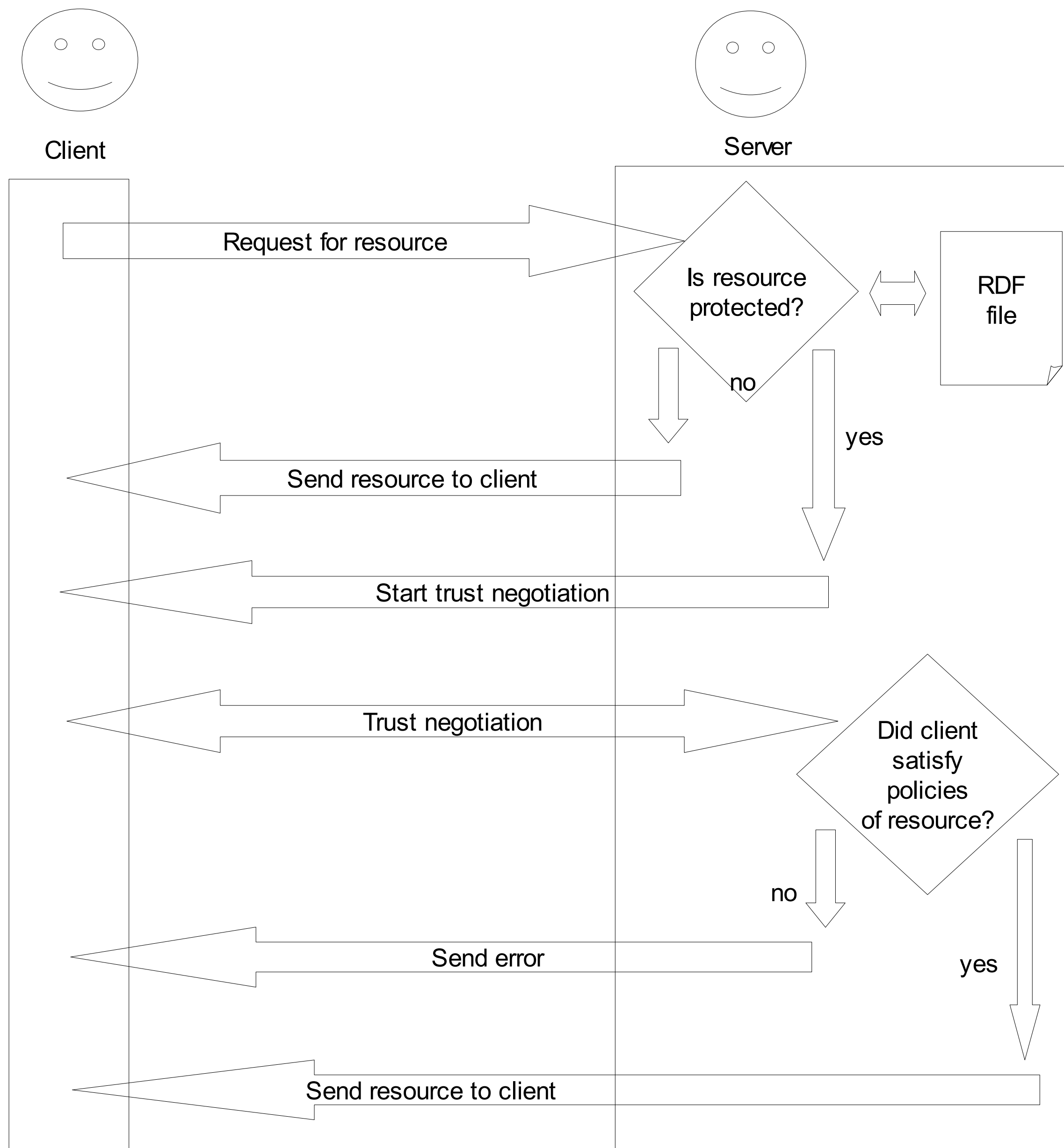
Static Protection **Dynamic Protection**

This chapter explains the concepts used and the next chapter goes into a more technical level.

5.1 Static Protection

When protecting static documents (as a whole), the person who runs a web server can perform this task by using the policy editor that was covered in the previous chapter. The policy assignments he performs there may be exported to an RDF [8] file.

The server infrastructure (with trust negotiation support) may now use this file to determine if a resource a client accesses is protected. If not, the content of the resource is sent back, otherwise a trust negotiation starts in which the client has to satisfy the policies assigned. If succeeded, he also gets the resource, otherwise an error message or explanation is sent back. The following diagram visualizes this.



The RDF file (which was generated by the policy assignment tool from the previous chapter) content is loaded by the server and the algorithm from section 4.3.2.1 is called to find out what policies (if any) are applied for the resource the client requests.

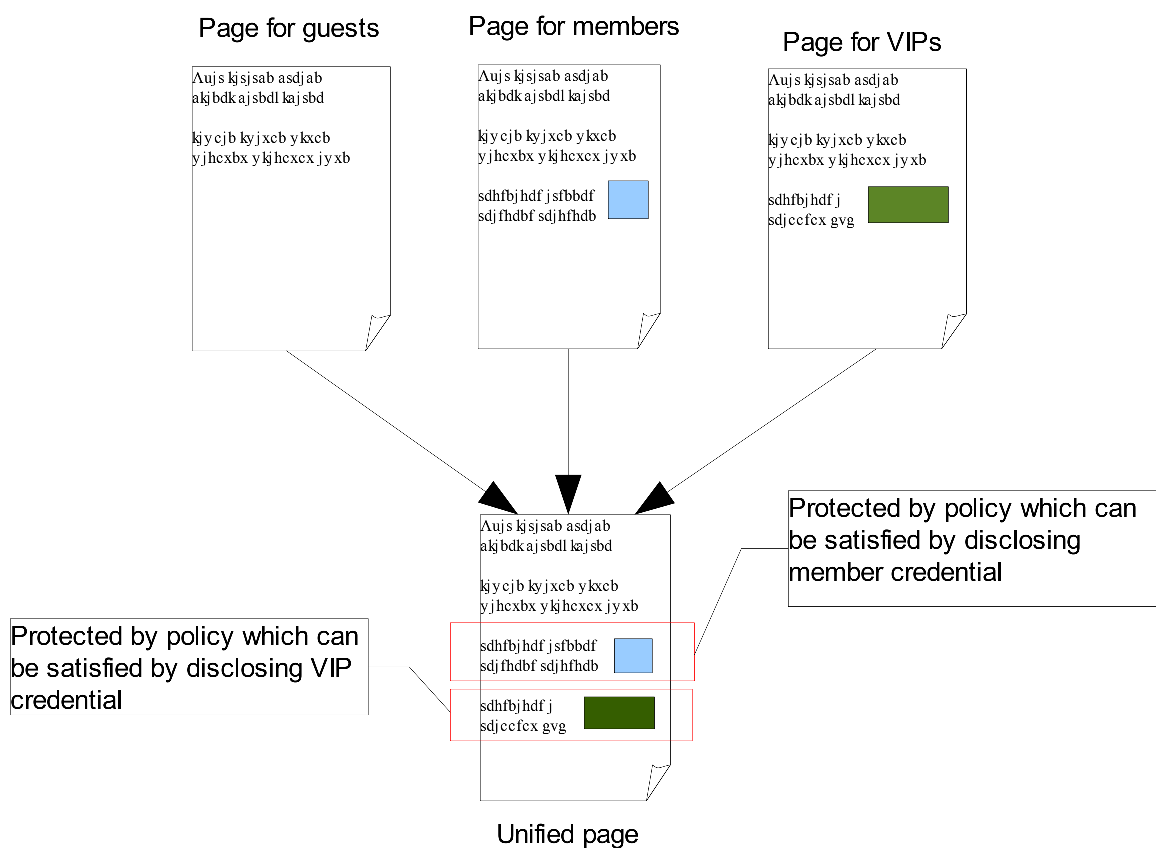
5.1.1 Example

An example RDF file is shown here (model was introduced in section 4.2.1.6). The Jena RDF library [13] was used for the implementation in this thesis.

```
<?xml version="1.0" encoding="windows-1252"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:peer="http://sourceforge.net/projects/peertrust/#" >

  <rdf:Description rdf:about="#_root_">
    <peer:startingPoint rdf:resource="#members.html"/>
    <peer:startingPoint rdf:resource="#paper.pdf"/>
    <rdf:type rdf:resource="#Root"/>
  </rdf:Description>

  <rdf:Description rdf:about="#members.html">
    <peer:policy>#Policy1</peer:policy>
    <rdf:type rdf:resource="#ProtectedResource"/>
  </rdf:Description>

  <rdf:Description rdf:about="#Policy1">
    <peer:default>true</peer:default>
       ...
    <rdf:type rdf:resource="#Policy"/>
  </rdf:Description>

  <rdf:Description rdf:about="#paper.pdf">
    <peer:policy>#Policy2</peer:policy>
    <rdf:type rdf:resource="#ProtectedResource"/>
  </rdf:Description>

  <rdf:Description rdf:about="#Policy2">
    <peer:default>true</peer:default>
       ...
    <rdf:type rdf:resource="#Policy"/>
  </rdf:Description>

</rdf:RDF>
```

5.2 Dynamic Protection

In contrast to the static protection just described which only allows to assign policies to the file as a whole, the dynamic protection focusses on the document level and allows the protection of certain content. If a person who runs a web server previously had multiple web page versions for different user groups (for example guests, members, VIPs) which share common content but had to be individually adapted for every user group, these pages may now be unified by protecting the relevant content that are exclusive for a special user group. This applies also for pages created automatically depending on the user.



Assigning policies to parts of a web document can not be done with the editor described in the previous chapter. Instead, the web document itself has to be manipulated in order to protect content. This document has to be a JSP file.

5.2.1 Java Server Pages

Java Server Pages (JSP) [14,18] is a technology developed by Sun whose main purpose is to generate dynamic web content for the client in HTML or XML (similar to PHP). JSP pages may contain special XML-like tags (which allow to include the desired functionality) and may also include Java code beside normal HTML- or XML-code. If a client now accesses the JSP page with his browser, instead of being sent right back (like an HTML file), the server transforms the JSP file into Java source code which is then compiled. The resulting server side program (a Servlet [15,18]) is then processed and generates the web content which is then returned to the client. So, the tags and Java code inside the JSP file control how this Servlet behaves and its output (which HTML passages should be written etc.). As additional advantage, the creator of an JSP does not need to learn Java, as he may use the functionality of the various already built-in tags. Instead of delving too much into details, only the relevant parts of JSP will be covered.

5.2.1.1 Custom Tags

A special feature of JSP can be used to achieve the intended result (protection of content of documents). Unlike HTML, JSP documents may use so called tag-libraries (which are archives of tags). They may be associated to a JSP page via a Tag Library Descriptor (TLD)-XML file. Having included the library this way, the tags contained in it may be used. Important to note is that custom tags [17,18] can be implemented and collected in TLDs. This way, the developer may enhance the capabilities of JSP by using his own tags or from a third party.

Coming back to the initial problem, the conclusion that can be drawn is that a special tag can help to protect content of a JSP document. As there is no built-in JSP tag that provides the intended functionality (determine if a client has satisfied the policies that were assigned to specific content and if yes, grant access), such a tag had to implemented for this thesis. Tags allow to use complex functionality in

an easy way. Everyone is able to use tags in his JSP documents without having to understand the exact implementation.

5.2.1.2 The policycondition Tag

The JSP tag that was implemented in this thesis for protecting parts can be used as follows:

First, the TLD (Tag Library Descriptor)-file has be associated to the JSP file:

```
<%@ taglib uri="policytag" prefix="poljsp" %>
```

- policytag is the TLD file
- poljsp is the namespace of the tags contained in the TLD file

Normal HTML-code can be used in the JSP file. The relevant parts which should be protected by a policy by using the custom policycondition tag:

```
<poljsp:policycondition policyname="<policyname>">
     <poljsp:iftrue>

          ...

     </poljsp:iftrue>
     <poljsp:iffalse>

          ...

     </poljsp:iffalse>
</poljsp:policycondition>
```

- <policyname> stands for the policy that should protect the part.

The content (HTML code) that a client sees if he satisfied the assigned policy should be enclosed in the „iftrue“-tag. In case of failure (client is not granted access), the „iffalse“ tag can be used (for error messages, hints why he can not see the protected content or just no HTML code). This way, content of web pages can be easily protected and assigned with policies, as the following example illustrates.

5.2.1.3 Example

A fictional soccer web page:

```
<%@ taglib uri="policytag" prefix="poljsp" %>
<html>
      <head>
            <title>Soccer page</title>
      </head>
      <body>
            Click <a href=“mail.html“>here</a> if you want to get
            the latest soccer results per eMail.<br><br>
            <poljsp:policycondition policyname="member_policy">
                  <poljsp:iftrue>
                        <applet code="LiveScores.class"
                                    archive="LiveScores.jar"
                                          height="400" width="100">
                              Activate Java in your Browser to see
                              live scores.
                        </applet>
                  </poljsp:iftrue>
                  <poljsp:iffalse>
                        Become a <a href=“join.html“>member</a> to
                        see soccer results live.
                  </poljsp:iffalse>
            </poljsp:policycondition>
            <br><br>
            <poljsp:policycondition
                  policyname="premium_member_policy">
                  <poljsp:iftrue>
                        Click <a href=“win.html“>here</a> to win a
                        soccer ticket!
                  </poljsp:iftrue>
```

```
                <poljsp:iffalse>
                </poljsp:iffalse>
            </poljsp:policycondition>
        </body>
</html>
```

5.3 Implementation

As already described, custom JSP Tags may be used by an TLD descriptor file. It describes the tags contained in it (their behaviour, name and the the location of their implementation on the server, attributes of the tags etc.). An extract of the custom TLD file which contains the policycondition tag just described can be seen in the following picture.

```
<tag>
	<name>policycondition</name>
	<tagclass>peertrust.tag.PolicyConditionTag</tagclass>
	<bodycontent>JSP</bodycontent>
	<info>Checks if one policy is satisfied</info>
	<attribute>
		<name>policyname</name>
		<required>true</required>
	</attribute>
</tag>
```

The implementation of custom JSP tags can be done in two different ways, the easiest one is to do it declaratively by using JSP tags. This solution does not require any Java knowledge at all and is ideal for beginners, the drawback is that the existing tags may not be sufficient to realize what the developer has in mind. The other alternative (that is also used in this thesis) is to use Java for the implementation of the tags behaviour. The Servlet API (not included in the standard Java JDK, is provided by external JARs by Tomcat for example) can be used for that. It allows to create so called Tag Handlers [17,18], classes which may be extended to implement the tasks the tag should perform. These classes act as a listener, as special methods are called from the server automatically if a client requests a jsp file that contains the corresponding tag. The <tagClass>-tag from the example TLD file above hints the server about where to find the tag handler class.

The tag handler allows easy access to the attributes and their values that were

written in the calling JSP file. In the case of the policycondition tag, the policyname attribute which specifies the policy that is assigned to the part is of interest here. While tag handlers offer a lot of opportunities (generating additional content etc.), the one implemented for the policycondition tag just checks if the client has satisfied the policy which was given as an attribute. If yes, the content that is enclosed by the <iftrue> tag is added to the output that the Servlet (generated from the JSP page) sends back to the client, otherwise the one from the <iffalse> tag. For these two tags, special handler classes exist as well which cooperate with the one from the policycondition tag.

Important to note is that if the user requests a JSP page, for each tag the appropriate handler class has to be called two times (instead of traditional tag handlers which only get called one time). The reason is that as a first step, the policies of all the tags on the JSP page have to be extracted to see which requirements the client has to satisfy for each tag. The second time is done after the following trust negotiation, here it is checked if the client satisfies the individual tags on the page or not. If yes, the content enclosed by the <iftrue>-tag is generated for the client, otherwise the <iffalse>-enclosed part. This is different compared to the static protection of resources in which an error page is displayed if trust negotiation fails.

A web designer who plans using such dynamic protection has to take into consideration that the handling described above results in more http messages than an access to an ordinary page (this applies to trust negotiation over http in general but even more to JSPs containing the tag described above). This also depends on the amount of policycontition tags used in a JSP.

The JSP handling at a more technical level will be covered also in the next chapter as it deals with the client and server architecture. This applies also for the implementation for the static protection.

6 Policy-Driven Negotiations on the Web

To adapt trust negotiation for the World Wide Web, certain modifications have to be made. Server and client side have to be enhanced to support trust negotiation by software agents that run in the background and perform the negotiation automatically. This chapter will focus on the client and server side architecture and implementation that were developed in this master thesis.

6.1 The Client

If a client wants to access a resource (for instance a web page) in the World Wide Web that is protected by a policy, a trust negotiation (as already described) must be performed in order to be granted access. This is done by a software agent that runs in the background and must be provided externally, as shipped browsers do not contain such a feature.

An ideal solution would make the installation of the software intransparent to the user (so that he does not recognize it), as they might quickly lose interest if they are required to manually install such a software by themselves. As a great variety of different browsers exist (Firefox, Internet Explorer, Opera, Safari, etc.), the trust negotiation software agent should support all (or at least the most important of them). Many browsers can be extended by plugins which may also be user defined, but this approach is not adequate as no common standard exists and providing individual versions for each browser will complicate things too much. Using a common language that most of the browsers understand to implement the software would be the preferred approach.

As a further prerequisite, the software must be able to intercept the browsing behaviour of the client in order to perform the trust negotiation. If a user clicks on a link to a protected resource or data is returned from the server, the software

should detect this in order to act accordingly (the trust negotiation messages between client and server are only relevant for the agent and should not be seen by the user).

6.1.1 Implementation

The Java language [9] was used for implementing this software. Java is platform independent and supported by a majority of browsers which allow Java applications to be easily integrated into web pages (via Applets or Java Web Start). This has a deep impact on the World Wide Web, as a web designer can use Java to enhance his web content (games, customized menu bars, tickers etc.) with more or less complex programs.

If a user accesses a web page which contains those, they will be downloaded on the clients computer and executed on it locally if Java is enabled and installed (so the server can save processing resources). In order to take security issues into account and avoid misuse, Applets for instance have only limited abilities per default (sandbox model). Security-related capabilities (reading and writing files of the client, establishing a http-connection to another domain etc.) have to be granted by the user or the applet has be signed by a trusted third party (Verisign, Thawte etc.) which the clients browser accepts, otherwise he may ask the user if this applet should be executed. If properly signed, the user would not recognize the installation process.

6.1.2 Applets

Recapitulating the above requirements of the trust negotiation software (support of popular browsers, one common standard/language, installation not visible), Java applets seem to be an appropriate option which was also used for the implementation developed in this thesis.

Applets need to be placed somewhere in a web page, as they have a visible content (whose width and length may be specified), so in order to hide them from the user (as the trust negotiation software should run automatically in the background), it may be placed in a zero-width frame (this also guarantees that the applet is loaded only once and not at every invocation of a link).

6.1.3 Java Script

For allowing the trust negotiation software to effectively take place, the applet has to communicate with the web page. For instance each link a user clicks on has to be recognized by the applet as the http connection to the target has to taken over by the applet for enabling trust negotiation (this will be covered later in more detail). As Java itself is not capable of communicate with the web page it contains, another language is needed to fulfil this task (Java Script for example).

Java Script [19] is an object oriented script language (developed by Sun) that can be used in web pages of the World Wide Web to include capabilities beyond the scope of HTML. Various functions exists which offer comprehensive possibilities (manipulating web pages and its items, validating user entries, reading and writing cookies for the client) which also allow misuse (unasked opening of a new browser window – popups – or unrequested closing of the browser). The greatest disadvantage is that Java Script is not integrated in the widely used Microsoft Internet Explorer which instead offers JScript [20] which is similar but unfortunately not fully compatible to JavaScript. As it is easy to find out which browser the client uses programmatically, code which supports both script languages can be written, this task is nevertheless quite cumbersome, as the JScript equivalent to a Java Script function has to be always detected.

Another two features which Java Script/JScript offers and which are important for the trust negotiation applet and its communication with the web page containing it are event handling and the possibility to access methods in applets). Event

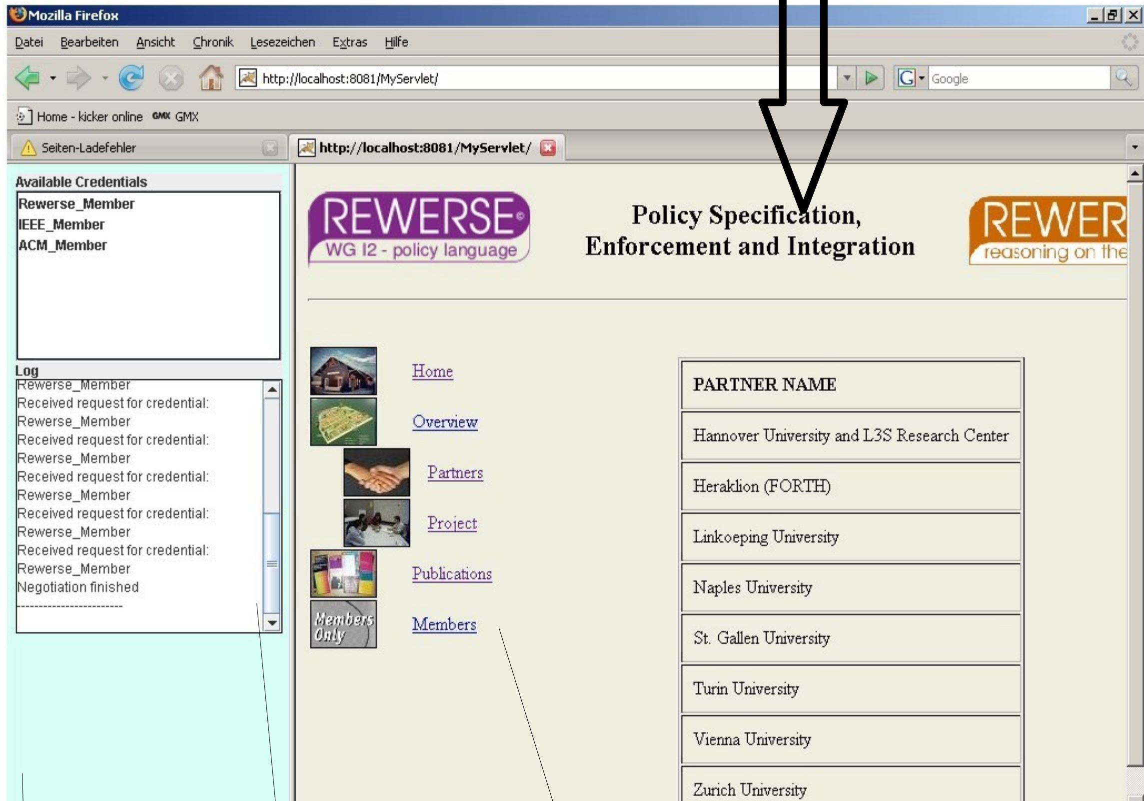
handling allows to detect various events (user opens a new page, closes one, resizes one, clicks on a item of the page, mouse cursor is on an web item etc.) and act accordingly. The requirements mentioned above that the applet has to be notified of link activations from the user can now be realized.

An event handler that detects if a user activated a link can be attached to all link items on a particular page and notify the applet (as JavaScript /JScript can call its methods) in case of action. This was implemented in a special script file that can be used for web pages which want to use the trust negotiation implementation of this thesis. An overview figure is shown below. A variable may specify links from which domain are affected (as outer links can't be protected anyway).

Java script file

```
If page is loaded the first time
       init()

function init()
       - attach an event listener to
       all link items on the page,
       so that if a user clicks on a
       link, linkevent() is called

function linkevent(event)
       - if domain of link is specified as variable
              - show wait screen
              - access applet method to
              perform trust negotiation and
              show the result
              - prevent default behaviour of
              link activation
```

Include script into web page so that all link items behave and contact the applet as specified in the script file



The included script file changes the behaviour of the link items, they contact the applet as specified above

Frame containing the applet, should have a zero-width, this is not the case here because of testing purposes

Trust negotiation applet for the client side that runs in the background, is visible here because of testing purposes

6.1.4 Applet - Mediator between Client and Server

One task of this thesis consisted of making trust negotiation in the World Wide Web possible. The L3S Research Center [3] does research on trust negotiation since many years and has already developed different powerful parts like engines which perform the trust negotiation. This trust negotiation engines are simply used in the applet as developing a custom one is out of this thesis scope.

The PeerTrust framework [4] (by L3S) allows easy configuration and extension of individual components. This is done by the use of interfaces, abstract classes and special configuration files which offers the choice between configuring the system hard-coded or declaratively. Using the latter option supports Java Reflections, so it is possible to integrate or replace a component into the program without having to recompile. Adapting trust negotiation for other environments can be done easily with this approach.

6.1.4.1 Communication Channel

One example of such a configurable component is the communication channel that the engine uses. Without delving too much into details, such a communication channel provides methods for sending and receiving messages (special classes). Additionally it contains information on how to contact the recipient precisely (address, etc.). This way it is easy to adapt the engine to work with any communication method or protocol as seen later.

The current implementation of the PeerTrust framework only provides socket communication which is efficient, persistent and bidirectional and but can not be used for the World Wide Web, as firewalls on either the client or server side may block all ports except the ones used for standard WWW protocols like HTTP, FTP, POP3 etc. A possible approach to use trust negotiation on the World Wide Web

and with browsers, is to employ HTTP (Hypertext Transfer Protocol, the basic WWW protocol). The benefit of ubiquity (all servers and browsers understand it) comes with the price of being stateless (in contrast to the persistent socket communication). This problem has to be taken into account and will be reviewed again later.

Providing an HTTP implementation for the communication channel is more or less straight forward, the HttpUrlConnection class from the Java JDK can be used for that. It allows to use the different HTTP modes (GET, PUT, POST, etc.) when requesting the server and get the returned response. The method for sending messages establishes an HTTP connection and send it via the POST mode. The method for receiving messages has to wait until the response is received and parse the contained object. These two methods are used by the policy engine which performs all the negotiation automatically. Their equivalent on the server side will be covered later.

6.1.4.2 Waiting Screen

If a user requests a resource, the duration of trust negotiation may take more or less time depending on the policies involved and on the desired object itself (it may be a JSP file that contains a lot of policycondition tags which were covered in the previous chapter), not to forget the bandwidth and processing power of the the clients computer.

As these factors may affect the time the user has to wait until the result after trust negotiation has finished, if this lasts too long, people may quickly become impacient or think that the browser has crashed. In order to assure them that the browser is still working and to make the progress more responsive, a popup wait screen is shown when negotiation starts and closed after it finished. This is done via JavaScript commands. Another approach (which employs JavaScript via the applet) that tries to use the existing page for a wait message instead of a popup

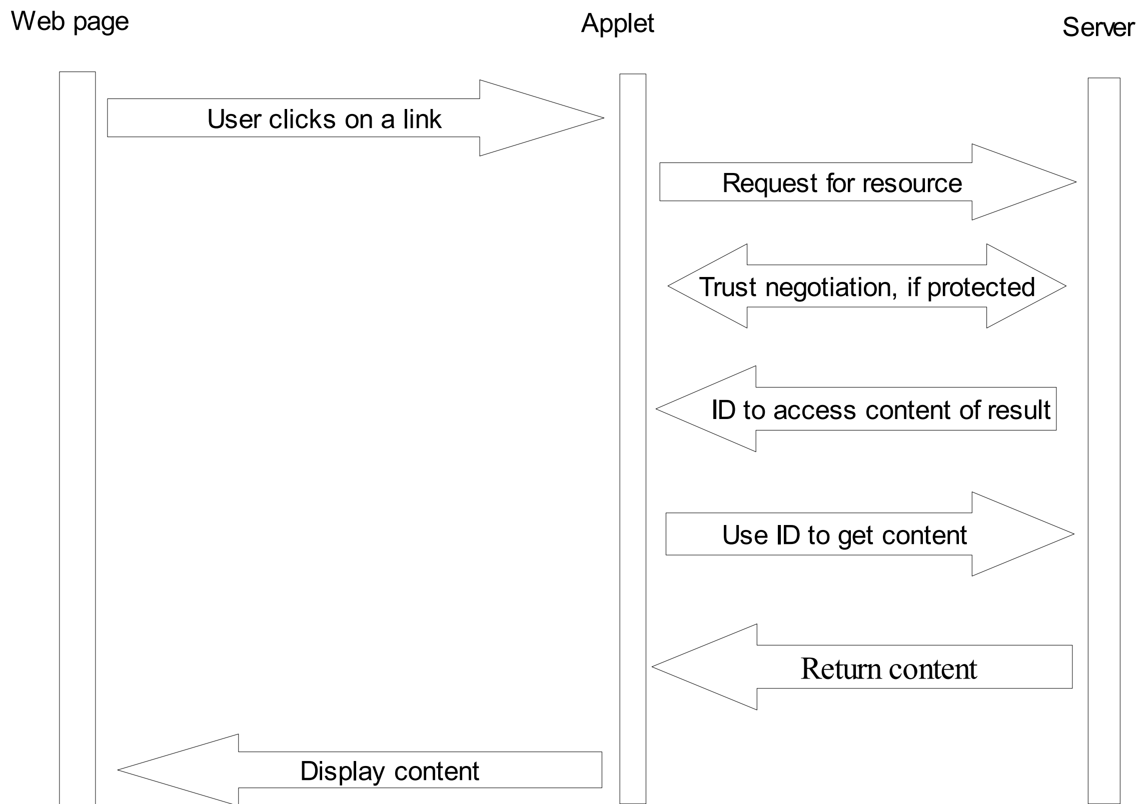
failed, because the wait page has to be deleted from the browser history after trust negotiation finishes (as the user should not be able to see it if he clicks the back-button of his browser). While this deletion from the browser history worked fine in Firefox, it did not work in the Microsoft Internet Explorer and no equivalent JScript command could be found, so the popup approach was kept to show the wait screen.

The existing wait screen may be extended in the future by displaying messages to the user what is currently going on. This may include policies that have to be satisfied or that are sent, credentials sent or received or explanations why a trust negotiation has failed and advices how to let them succeed the next time.

6.1.4.3 Displaying the Result

If trust negotiation finishes, a result should be displayed for the user. This may be an error page (and advice) if it failed or in case of success the content of the requested resource. In both cases it is the task of the applet to obtain and display the result. That is because the users click on the link pointing to the resource was intercepted by the applet as described above. It contacts the server to find out if the resource is protected or not. In the first case, multiple rounds of trust negotiation may follow. After that (and also in the second case), the server has stored the content of the result exclusively for the client and so he sends back a special ID (the exact mechanism will be covered later in the description of the server implementation). The client uses this ID again as a parameter for his final request for this content the server has reserved for him. This is necessary because the client cannot generate and display a web page himself (the document.write method in JavaScript works only for text files, not for binary data), so the content has to be delivered by the server in such a way that only the client and no other one can get access. As the applet has taken ownership of the HTTP request, the server sends the answer back to him (and not to the web page like in the traditional case), so at the end of trust negotiation it is the task of the applet to display the requested resource (or the error page), as he hindered direct communication between server and web page to take place. The following

diagram illustrated the behaviour desribed. The server part will be covered later in more detail.



Instead of applying this behaviour, the JavaScript file may be modified that only links which point to a specific domain call the applet (via a variable for instance), this way the web designer can reduce traffic, by applying the costly trust negotiation (in term of http messages exchanged) only to links which really need it.

The applet itself reads the RDF file that was created with the policy management tool when it is loaded the first time and applies the relevant information to the policy engine. As the applet needs to know the exact location of the file on the hard disk of the client in advance, it expects it to be in the folder that represnts the „user.dir“ Java system property. The applet is signed and given the permission to access this directory.

6.2 The Server

Another task of this master thesis was to design and implement a server software which is capable of performing trust negotiation with the client applet that was just described. While the client implementation task is to intercept the default link behaviour and to display the resource content or error page on the web page (depending on the result of the trust negotiation), the server has to watch on the protected resources on it and send their content to a client requesting one only if the trust negotiation succeeds and the policy is satisfied. Otherwise an error page is used instead. The traditional server behaviour is not sufficient here, so it has to be extended. While the previous section introduced the client perspective of trust negotiation and treated the server as a sort of black box, this gap will be filled now.

The implementation should allow the web designer to protect his web content easily, so it loads the RDF file from the policy management tool at server startup. Nevertheless he has to take the previous section into account, as also the applet must be provided and the JavaScript file which manipulates the default link behaviour has to be included into every web page that has a link to a protected resource. No further configuration of the server implementation is needed, so that the web designer does not need to be an expert in order to protect his resources and to require the clients to use trust negotiation in order to access them.

Another complex task of this thesis is the dynamic protection of resources. This is done with JSP files which contain the policycondition tag, as already described in the previous chapter. The approach of dynamic protection will be investigated in this section in a more technical level.

6.2.1 Implementation

For this thesis, the Apache Tomcat Server [21] has been used for implementation and testing. In order to support also other kind of servers, the Servlet technology [15] has been used.

6.2.1.1 Servlets

The web content that is located on servers typically consist of static documents (like HTML, PDF, images, etc.) and dynamic web pages. Before Java was popular, CGI (Common Gateway Interface) was widely used to create them. The main shortcomings of this approach were platform dependence (server migration may result in adapting the existing code) and the fact, that it did not scale well (performance dropped when accessed by multiple clients). Programming skills in C, Perl and knowledge of the HTTP protocol were necessary. Since 1996 companies began to use Java on the server side to address those disadvantages, different solutions that were tied to individual servers were united 1997 by JavaSoft by the finalization of the Servlet [15,18] technology which defines a common standard that a wide range of server support. As the implementation used for this thesis uses this standard and no Apache Tomcat specific components, it may also run on other servers.

Servlets are Java programs that run on the server side and may manipulate the HTTP request and response. This can be used to generate dynamic web content, as binary or text data may be assigned to the response pro grammatically. As already covered in the previous chapter, Servlets may be written in Java itself or as JSP pages [14,18] (an approach that is perfect for beginners, as no Java knowledge is needed) which are compiled automatically into a Servlet if a client accesses such a page.

Like in RCP (chapter 5), a Servlet may be configured by an XML file declaratively, this will be seen later in more detail.

6.2.1.2 Filters

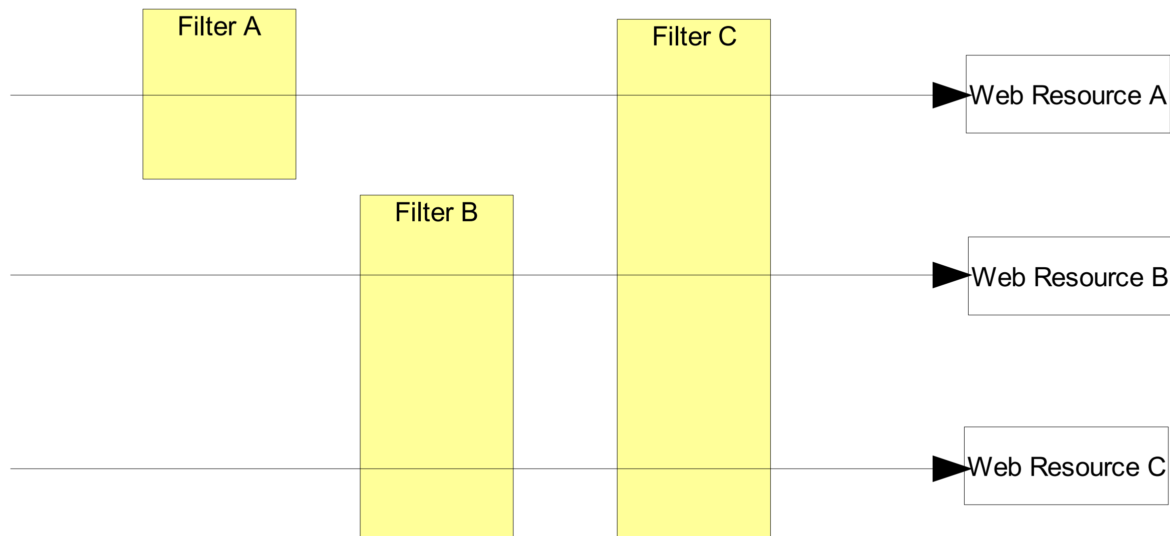
Besides custom JSP tags (via tag handler classes, as described in the previous chapter), the Servlet API offers another feature that fits perfectly for the server side implementation of the trust negotiation agent: filters [18].

Filters may be applied to a set of web resources resources and allow to control and modify the HTTP requests from and responses to a client. Filters may be used to

- Check the HTTP request to a web resource or the response sent back
- Modify their headers and body data
- Block them so that they can not reach their destination
- Interact with external resources.

This mechanism may be used to implement authentication, logging, data conversion and compression, en- or decryption, etc.

The use of filters is not restricted, as a web resource may be affected by an arbitrary number of filters (a so-called filter chain whose order may also be specified).



The trust negotiation agent server side implementation for this thesis mainly consists of two customized filters

- Filter for static protection of resources
- Filter for dynamic protection of resources

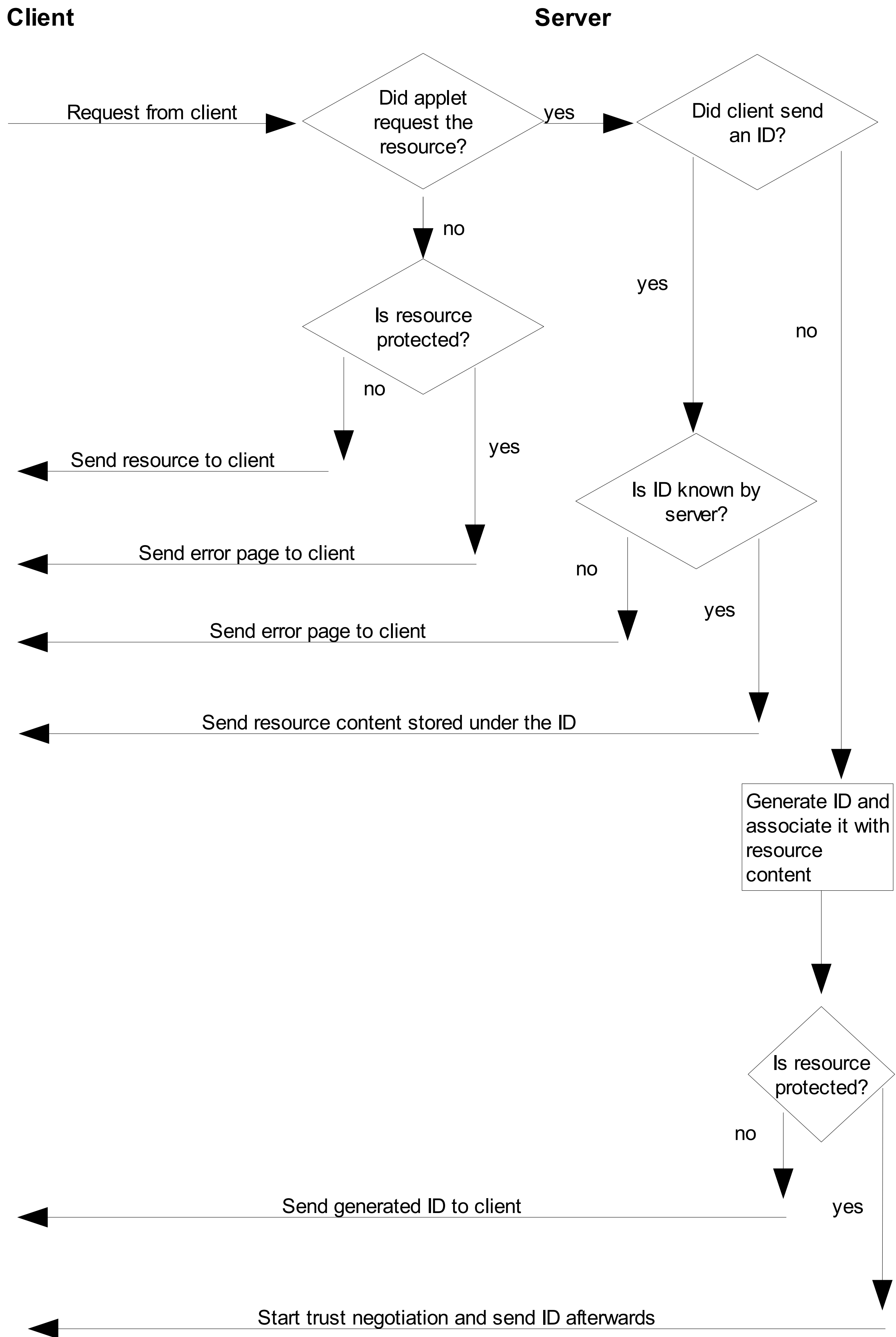
6.2.1.3 Static Protection Filter

This filter is applied to all web resources, so every request from a client can be checked and it can be determined if access to the resource is granted or not. When the Servlet is initialized, the filter loads the RDF file that was made with the policy management tool that was introduced in chapter 4. The main task of this filter is to check if the requested resource is protected, this can be determined with the RDF file and the policy retrieval algorithm for a given resource (see section 4.1.2). If this is the case (and an applet accessed the page), a trust negotiation is started in which the client has to satisfy the protecting policy, otherwise the server stores the resource content and sends back the ID which the client can use with his next request to get the resource (see previous section about the client implementation). This is done also if trust negotiation is finished.

A distinguishment has to be made between applet and normal browser request for

a resource. As the applet intercepts the link activation, it is responsible for displaying the target of the link (document) and an additional request (in which the ID given by the server is included) to the resource has to be made (as described in the previous chapter) after trust negotiation. This behaviour does not suit well for normal browser access (for example the user might access a resource via the browsers address field instead of clicking a link). Instead of sending back an ID (like applet requests are handled), the resource content (if not protected) or an error page (if protected) is returned by the server. The server can distinguish between applet and browser access by checking if a special URL parameter is included.

The following diagram will illustrate the filters behaviour. Trust negotiation messages are not considered here (they will be directly passed to the negotiation engine).


Client
Server
Request from client
Did applet request the resource?
yes
Did client send an ID?
no
Is resource protected?
yes
no
no
Send resource to client
yes
Is ID known by server?
Send error page to client
no
yes
Send error page to client
Send resource content stored under the ID
Generate ID and associate it with resource content
Is resource protected?
no
Send generated ID to client
yes
Start trust negotiation and send ID afterwards

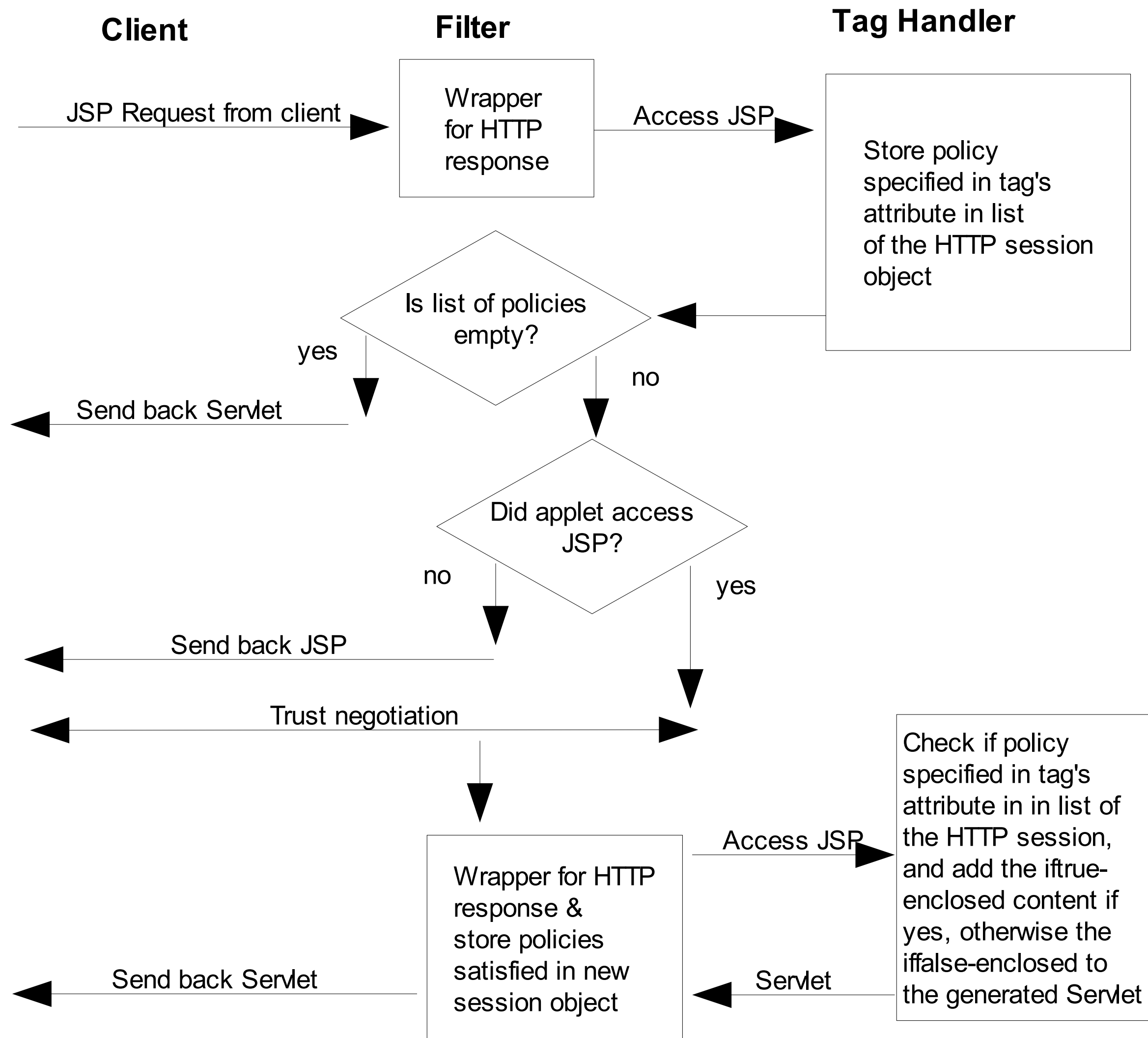
6.2.1.4 Dynamic Protection Filter

As dynamic protection is done by using the policycondition JSP tag (as discussed in the previous chapter), the filter for dynamic protection is only applied to JSP files. The handling of the custom tag takes place here, a short introduction was already given in the last chapter, this section delves more into detail.

If a request for JSP file comes in, the documents content has to be investigated first to check if it contains the policycondition tag and additionally extract the associated policies which are specified as the tags attributes. To do so, a special wrapper class is used that allows to access the resource with a wrapped (faked) HTTP response (as using the original response would directly send the resource to the client) which can be checked by the filter afterwards and may either be forwarded to the client or not. The filter uses this wrapper to find out all the policies of the policycondition tags in the JSP (if available). For each appearance of such a tag, the corresponding tag handler class is invoked. As the wrapper invokes the invocations of the handlers, the policies have to be sent back to the filter somehow.

Because this can not be done with the wrapped HTTP response, an HTTP session object (which maintain state between multiple requests from a client, these are secure, because these are managed on the server side, the client is not able to access them) is used to store the policies. So each invocation of the tag handler class adds the corresponding policy to the list contained in the session. The filter retrieves the policy list from the session after the wrapper has finished. If this list is empty, the wrapped HTTP response is forwarded directly to the client, because no policycondition tag is contained in the requested JSP file. Otherwise, a trust negotiation starts with the client in which he may try to satisfy the policies. In either case the session object is destroyed, because it would not be used any more). If the client did not request the JSP page with the applet (detected by the presence or absence of a special parameter), an appropriate error page is sent to the client that advices him to use the applet.

In contrast to the static protection, the negotiation may fail, the client will be able to see the JSP page even if no policy was successful, the policies which he satisfied or not will only contribute to the visible content. So the JSP page and the tag handlers have to be accessed a second time in order to check which policycondition tags contribute to the generation of the page which the client will see afterwards. To do so, the filter uses the wrapper again, after creating an HTTP session object and adding the policies which the client satisfied to it. Instead of retrieving the policy tag attribute as described above, the tag handler now checks if it is contained in the policy list of the session. If yes, the tag handler adds the content enclosed by the „iftrue“ tag to the generation of the page/Servlet, otherwise the one enclosed by „iffalse“ (see description of policycondition, iftrue and iffalse tags in the previous chapter). The session is again destroyed afterwards.



The handling of a dynamic page that contains the policycondition tag is requires more HTTP requests and responses compared to a static resource. Additionally, the static protection filter is also affected by JSP pages, so it is possible to protect the JPS page as a whole (statically) and contents of it via the policycondition tag at the same time.

7 Related Work

XACML [22] is a access control profile and the policy assignment approach is quite similar to the one developed for this thesis. Policy inheritance in a hierarchy of resources is also supported. While this thesis concentrate on file and credential hierarchies, this profile is more general, as arbitrary hierarchies of resources may be used (for example also nodes in a XML document may be protected). This paper generally deals with how resources in a hierarchy or requests to them may be specified (a request to a node in an XML file is made via an XPath query). Policies are not directly assigned to a specific resource (as done in this thesis), instead they are specified separately and assigned to the resources via XPath or regular expressions.

[23] suggests a new way a client can use to establish trust to a server. Usually (also in this thesis) the client may only react to the server. If the latter one asks for the disclosure of a credential, the client may send a policy in case of protection, but he can not take the initiative, after the client makes the request, the server guides the negotiation, not the other way round. The new approach however addresses this shortcoming by allowing the client to start a trust negotiation with the server before he requests the resource. This allows the client to be certain that the server is trustworthy and provides the desired capabilities (for example if he handles private data appropriate or supports certain security mechanisms) before accessing it. A novelty is also the fact that the content a client is required to disclose is checked (by using predefined rules, text patterns, algebraic or probabilistic models, the disadvantage is that misclassification may occur) if should be protected or not (content classification). If yes, a policy is automatically generated (not assigned statically like in the traditional approach) and send to the server (dynamic policy association). TrustBuilder (existing trust negotiation agent) was extended prototypes of the new technique which work also with HTTP (and additionally SMTP). In contrast to this thesis, trust negotiations is performed by the use of new HTTP headers in messages here. Unlike this thesis which used additional software on the client side (the applet) to support trust negotiation, this

approach uses a proxy server which provides all the functionality but lacks the clients ability of fine-grained control over his sensitive content/resources. In contrast to the thesis, the technique concentrates on data that the client sends to the server (via a web form for example), not on static protection. Content triggered protection is the keyword.

[24] also deals with trust negotiation on the World Wide Web. A proxy server is also used to realize the client side implementation (unlike this thesis which uses an applet for the browser) and perform the trust negotiation for the user. A rich infrastructure has to be provided, as beside the proxy additional fact-servers are also required which help to authenticate the user by cryptographical keys and store in this cases the requirements a client has to fulfil in order to gain access to a resource. The server part however is quite similar to the one used in this thesis, as also a Servlet (which uses components for negotiation and policy discovery) was developed that can be easily installed on any Servlet-supporting server. Also the fact servers can be realized by using an existing Servlet. Instead of using policies and credentials, the server generates a proposition which the client has to prove logically and cryptographically (in cooperation with the fact server) in order to gain access to the requested resource. This results in a lower number of messages that have to be exchanged to establish trust.

The policy inheritance (mandatory and default policies) proposed in [5] was adapted for this thesis, however new features like filters and exceptions have been introduced. This paper also investigates possibilities to support the user in assigning policies to resources. The policy management tool PolicyTab (a plugin for the ontology editor Protégé) was a great inspiration for the counterpart implemented in this thesis, both programs are quite similar (the plugin offers more features, while the tool of the thesis is a stand alone product). In contrast to the thesis, this paper does not take dynamic protection of content into account.

[6] presents a suite of four tools that should assist the user in trust negotiation over the SULTAN trust management model. The specification editor is similar to the

policy management tool implemented in this thesis (assigning and managing policies), additionally it offers to specify new policies. The policies are written in a Prolog syntax, so an analysis tool is provided which can be used check the consistency of policies, properties etc. and to query them in order to find potential conflicts or cycles. The risk service tool allows to calculate the probability that a trust negotiation for a sensitive resource fails, the monitoring service tool notifies an administrator of of potential conflicts and risks the next time he uses the specification or analysis tool. Although these tools really help web designers, they are aimed to more experienced users, not to beginners, as Prolog skills are required.

[1] introduces PeerTrust whose trust negotiation framework and policy engine was also used in this thesis. The paper also proposes using an applet in the browser but in contrast to the counterpart of this thesis it was not not so adequate for the World Wide Web as the applet communicated to other peers over sockets which may cause trouble with activated firewalls. JavaScript support to communicate with the web page containing it is missing, also the dynamic protection of resources.

8 Conclusions & Open Issues

Trust negotiation is a technique that can be used to replace the traditional registering/login approach. Although it is still widely used in the World Wide Web, its shortcomings (client may have no chance to find out if server can be trusted, he may not choose which personal information should be disclosed, verifying customer data is hard for the server, user needs to remind the password, may choose a stupid one which can easily be guessed or they get tricked into revealing it) are addressed by trust negotiation. This approach offers every party (client, server etc.) to protect his resources (documents, files, credentials) by assigning policies to them which tell the other party what to do in order to be allowed to access the resource (for example by disclosing special credentials). As such a negotiation may be complex and may last multiple rounds (if a client wants to access a resource from the server which is protected by the policy which requires the disclosure of a credential which the client has also protected etc.), it is not performed by the client himself (as this would be too tedious), instead this task is done by software agents transparently in the background.

While the approach is widely used and tested under closed peer-to-peer networks, this thesis focusses on the use in the World Wide Web in which clients may use existing browsers to access web resources which may be protected by the author. In order to become more popular, trust negotiation must be easy to employ, users should be able to take advantage of it with minimum requirements. Many non-expert-users may resign if assigning policies to a resource or writing policies is too complex. This also applies for the web designer who wants to integrate the trust negation support for his web page. Many novice users do not want to invest hours of learning how to use a new technology, so supporting and guiding them is a good idea to lower the barrier and attract their interest for trust negotiation.

This thesis focusses on the easy use of trust negotiation in the World Wide Web, for both clients and web designers. As a first step, a special platform-independent

tool was developed that assist users in assigning policies to files in order to protect them. A web designer may use it to easily protect his documents on the server. Another mode offers to protect credentials (a clients credit card for instance). A native graphical user interface assists the user which also visualizes the hierarchy of resources and inheritance of policies (a resource inherits all the policies from its parent, if not desired, a user may overwrite the unwanted policy). While the hierarchy of files is already given by the users hard disk, it can be modified for credentials by the use of abstract classes via drag and drop support. The resource locations and the assigned policies can be imported from and exported to an RDF file.

This file is needed for the trust negotiation agents on the client and server side which also had to be designed and implemented for this thesis. An existing trust negotiation framework named PeerTrust was used. As trust negotiation is not integrated into browsers (Firefox, Internet Explorer etc.) nowadays, this functionality has to be added for the client. As the trust negotiation agent should run in the background, an additional software was needed. Instead of requiring the user to manually download and install a program, an Java Applet was used which the clients browser overtakes this tasks transparently to the user. This applet is hidden in an extra frame and JavaScript (JScript for Microsoft Internet Explorer) is used for the adaption of the users browsing behaviour.

Besides needing to provide the applet and integrating it into the web content, a web developer must use an web application which was implemented for this thesis. It reads the resources and assigned policies from the RDF file. The web application may modify and control the HTTP requests from and responses to a client via a technique called filters. So the requested resource can be sent back if it is unprotected or trust negotiation was successful, otherwise the HTTP response is modified to display an error page instead. Not only whole documents, also specific content of one can be protected by the web designer. So the same page may look different to individual users (for instance users may only see specific text or images if they satisfied the policies protecting them). For the implementation, a custom JSP tag was implemented that allows the assignment of policies to specific

parts of a JSP file. Special tag handler classes exist in the web application that implements the intended behaviour (in this case the integration or exclusion of the enclosed part of the JSP file into the generated Servlet depending on failure or success of the trust negotiation). This web application respects to the Servlet standard, so it can be used on a variety of different servers and is delivered as an easy-to-install WAR file.

What is still lacking in this thesis are real credentials (may be integrated into X.509 certificates), as the provided policy engine currently does not support them (instead it only works with strings). The current communication is done with HTTP, so to respect security issues more, HTTPS would be better. A nicer integration of the wait screen (which may also contain a list of messages and hints for the client) would also be nice in order to enhance the overall usability and feedback for the user. Another possibility would be to make the policy assignment tool collaborative, so that multiple user can work with it at the same time.

9 References

[1] No Registration Needed:
How to Use Declarative Policies and Negotiation to Access Sensitive Resources on the Semantic Web
R. Gavriloaie, W. Nejdl, D. Olmedilla, K. E. Seamons, M. Winslett
1st European Semantic Web Symposium, Heraklion, Greece, May. 2004

[2] Driving and Monitoring Provisional Trust Negotiation with Metapolicies
P. Bonatti, D. Olmedilla
IEEE 6th International Workshop on Policies for Distributed Systems and Networks (POLICY 2005), Jun. 2005, Stockholm, Sweden

[3] L3S Research Center
http://www.l3s.de

[4] Lecture “Development of Secure Software” held in summer 2006
http://www.se.uni-hannover.de/lehre/2006sommer/ss2006_ess.php

[5] Ontology-Based Policy Specification and Management
D. Olmedilla, C. C. Zhang, M. Winslett, W. Nejdl
European Semantic Web Conference (ESWC 2005), May/Jun. 2005, Heraklion, Greece

[6] Trust Management Tools for Internet Applications
T. Grandison, M. Sloman
Proc 1st Int.Conference on Trust Management, May 2003, Crete, Springer LNCS 2692, pp 91-107

[7] REWERSE
Reasoning on the Web with Rules and Semantics
http://rewerse.net/

[8] Resource Description Framework
http://www.w3.org/RDF/

[9] Java Technology
http://java.sun.com/

[10] Rich Client Platform
http://www.eclipse.org/rcp/

[11] Eclipse
http://www.eclipse.org/

[12] Standard Widget Toolkit
http://www.eclipse.org/swt/

[13] Jena Semantic Web Framework
http://jena.sourceforge.net

[14] Java Server Pages
http://java.sun.com/products/jsp/

[15] Java Servlet Technology
http://java.sun.com/products/servlet/

[16] Eclipse Rich Client Platform
J. McAffer, J.-M. Lemieux
The Eclipse Series

[17] Creating Custom JSP Tag Libraries
http://java.sun.com/developer/Books/javaserverpages/cservletsjsp/chapter14.pdf

[18] The Java EE 5 Tutorial
http://java.sun.com/javaee/5/docs/tutorial/doc/JavaEETutorial.pdf

[19] JavaScript Technology
http://java.sun.com/javascript/

[20] JScript Technology
http://http://www.microsoft.com/jscript/

[21] Apache Tomcat Web Server
http://tomcat.apache.org/

[22] Hierarchical resource profile of XACML v2.0
OASIS Standard, 1 February 2005
http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-hier-profile-spec-os.pdf

[23] An Access Control Model for Dynamic Client-Side Content
A.Hess, K. E. Seamons
8th ACM Symposium on Access Control Models and Technologies, Como, Italy, June 2003

[24] A General and Flexible Access-Control System for the Web
L. Bauer, M. A. Schneider, E. W. Felten
Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, August 2002