A Design Pattern for Improving the Performances of a Distributed Access Control Mechanism

Emiliano Tramontana

Department of Mathematics and Informatics, University of Catania, Italy

tramontana@dmi.unict.it

Abstract. *In a distributed environment comprising several clients and services, user permissions need to be checked before accessing a service. Having a centralised authorisation component, granting or denying permissions for each requested service, can affect performances due to the big amount of runtime requests on such a component and to network-related delays. The timing of replies is also affected by the workload of servers, as well as the computational complexity of requested services. This paper describes a design pattern that reduces delays by introducing a component that provides: (i) a local cache for permissions; (ii) the outputs of services for a custom validity time; and (iii) asynchronous service invocations to prevent clients from blocking their execution.*

Categories and Subject Descriptors

•Software Engineering Software Architectures-Patterns

General Terms

Design

Keywords

Performance patterns, distributed applications, caching, security, control-access-list, policy enforcement

1. Introduction

For security reasons, services in a distributed environment have to be guarded to let only authorised users perform the desired activities, i.e. each user is restricted to use only certain services and in a specified way. It is often the case that fine-level access rules are implemented, and accordingly a request has to be filtered against the access control list for the requested service, before executing it. Given the multitude of users, clients, and services, authorisation filtering on a per-service-access basis could adversely affect performances, since remote requests due to authorisation incur into network delays and a large number of messages could overload the authorisation service.

This paper introduces design pattern *Local Controller* as a solution that caters for permission checks on the host holding the requested service, hence greatly reducing communications with a centralised authorisation component. For realising this filtering ability and to reduce delays brought by computationally costly services, the access control list and the service replies will be cached.

The proposed pattern should provide performance benefits to an industrial-level application where a considerable amount of user requests are expected. Moreover, this pattern lets the developer consider both the advantage given by service distribution and possible bottlenecks due to centralised security related components. By reasoning on the balance of such forces the developer can come up with an improved design and high quality systems [YWW]. Thanks to this design pattern we hope to have shed light on performance issues that can be dealt with in the early design phase rather than as a fix on a system under test.

2. Performance Aware Pattern for Distributed Access Control to Services

Name

Local Controller

Intent

Reduce the number of centralised authorisation checks and the overhead due to network delays and/or computationally costly services by caching both access list for services and service replies and by resorting to asynchronous calls.

Example

Generally, modern applications are organised according to a distributed environment and include security requirements, hence they consist of a number of services that are located and run on a server-side and several clients, which can request the execution of desired services to read data, provide updates, or process some task on their behalf, etc. Clients are often tailored to the several types of user devices, hence client applications are developed for each hand-held device platform supported, as well as for more traditional hosts. In this scenario, the abilities of clients can vary widely, and some clients are developed e.g. to have read-only accesses to the contents provided by the server-side, others can have more comprehensive functionalities.

For a distributed application including security requirements, the requests to execute services performed by a client-side to a server-side, before being accepted, have to be checked and possibly given clearance by a proper authorisation service. According to a fine-grained authorisation policy, each request will have to be checked against its clearance policy, and then

a corresponding service can be accessed, this in turn would let the user read/write data. In a typical distributed environment, a centralised access control component has been deployed on a certain host. A per-request authorisation check has to be performed even though the user has been initially authenticated and the application session is active. Therefore, a large number of requests can be received by the central authorisation service, possibly affecting its performance. Moreover, some services could be computationally costly and take a substantial processing time. Then, the client-side has to bear: (i) communication delays due to permission request-grant messages, as well as communication delays due to service request-reply messages; (ii) unresponsive services due to long processing time or server overload.

Context

Generally, a central authentication service is provided for all users and services handled by an organisation, and the client-side could consist of components running inside a web browser, e.g. as javascript code, which has been downloaded from an http server, in other cases it is a user-device platform-dependent application. Such distributed applications could follow the Model-View-Controller design pattern [POSA2] and are supported by a framework, such as e.g. Spring [Spring]. The client component playing the role *View* embeds invocations to a server-side, playing the role *Controller*, and then this invokes services. Given the large amount of desired functionalities, a corresponding large amount of useful services are deployed. As a result of components distribution, clients requesting or sending data to services are often unknown by such services. The distributed system comprising clients and services has to have a way to authorise each user request. Moreover, some deployed services could be computationally costly, e.g. a query on a large database, or a filter on some data. The considered system has to cope with network communications, possibly long service processing times, and host or network overload.

Problem

The requests performed by the client-side have to be authorised one-by-one and the related interaction with remote servers can introduce unwanted delays. I.e. each request has to be checked by a proper remote authorisation service that once given a service id and a user id can tell whether the request can be served, however a user can perceive a delay due to the additional messages realising the network-based authorisation checks. Moreover, once the request arrives to a computationally costly service, the client could perceive the server side as unresponsive. Services execution could even overload their hosting machine.

Forces

- The permissions are fine-grained, i.e. a permission is assigned to each combination user identity/service identity, and checks have to be performed on a per-request basis, this can be time consuming if remote communication is involved.
- The client request could have been forged, i.e. the client cannot be trusted, then each request has to be checked.
- Clients can perform many requests, each generating an authorisation check, possibly overloading the authorisation service.
- Clients can perform many request to multiple computationally costly services, overloading the host holding services should be avoided.
- Capability given to the client-side, such as service identities or access control information cannot be trusted, hence security checks and related access policies have to be handled on the server-side.

Solution

Create component *LocalController,* on the same host as the service, holding an extract of the access control list corresponding to the permissions for the local service, hence requests for the local service can be filtered without asking a centralised *AuthorisationService*. Moreover, *LocalController* determines whether to actually execute a requested service when a recent reply has been cached.

A *Client* request, on a given known service, will be intercepted to perform authentication and authorisation. Such checks are attempted by *LocalController* (on the same host as the requested service) and in some cases propagated to the centralised namesake services when the requesting client is unknown. In the latter case, *AuthorisationService* provides the list of permissions for the requesting user and all the services hosted on the same machine. Such a list is then stored by *LocalController* on an instance of *Permission*. Subsequent requests from the same user will be filtered by *LocalController*. By caching the permission list locally, delays due to communication with *AuthorisationService* is borne only once, instead of per single request.

A service replies to a given request is cached on *LocalController*, and its expiration time is determined according to the service being invoked. Hence, a client requesting the same service multiple times will retrieve data from *LocalController* and skip the service execution if within the validity time of the reply. Validity time is determined by the service, thus it goes according to the reply being given, e.g. an immutable reply or a statistically slow changing value would have a relatively long expiration time, and vice versa a fast adapting reply will be given a very short validity time.

Furthermore, calls to time-consuming services are performed asynchronously, i.e. such calls are non-blocking for the client, and then a call-back method is available to provide the needed reply (the result value will be handed to *LocalController*, and in turn to *Client*). Hence, *Client* performing asynchronous calls can continue its execution, exhibiting responsiveness. Thanks to asynchrony, *Client* can e.g. show results of the faster-replying services, inform the user with status updates, take further user inputs, etc.

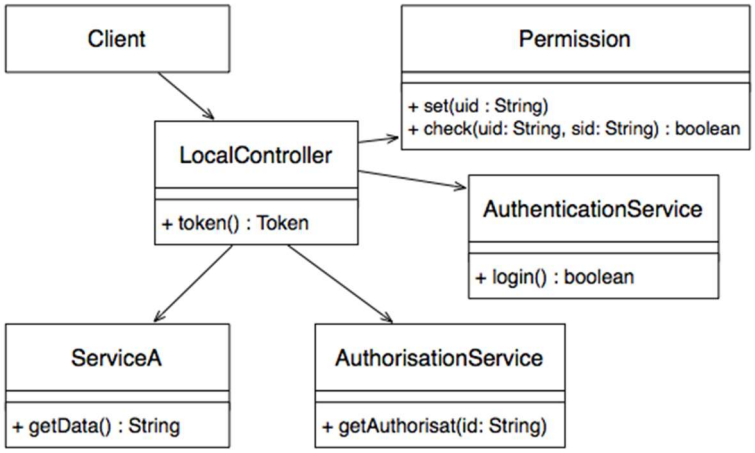

Figure 1. Class diagram for design pattern Local Controller

Structure

Figure 1 shows the dependencies between the said components composing the design pattern. A client request is firstly checked by a *LocalController*, for all the services located on the same host, and once its permission list has been cached, it checks whether access can be granted and then hands the request to the requested service.

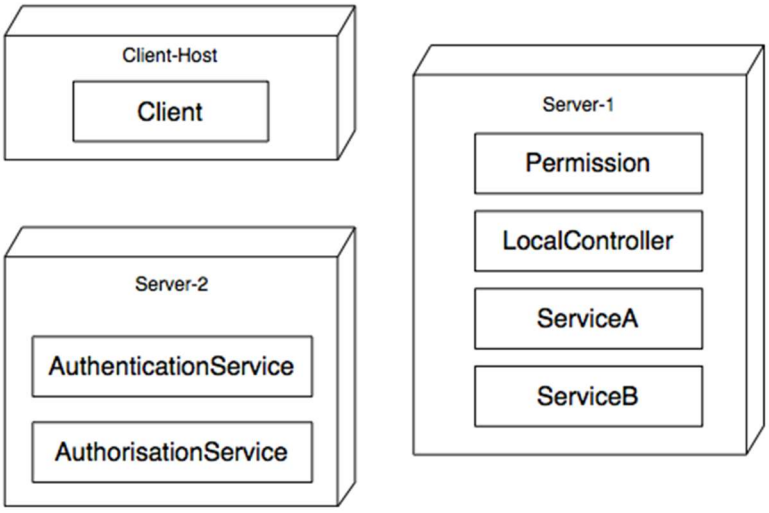

Figure 2. Deployment diagram

Deployment

Figure 2 shows how the involved components are spread among hosts. In such an example, the two available services, *ServiceA* and *ServiceB,* are held on host *Server-1* and a single *LocalController* performs authorisation checks and caches service replies. The other components, i.e. *AuthenticationService* and *AuthorisationService* are held on a different host. The general case would comprise other application services deployed on other hosts.

Participants

- Authentication Service
  - is a server side component that provides users a way to gain access to several services by holding and checking user credentials
- Authorisation Service
  - is a server side component that holds access rights to services that grants or denies permission according to the user identity and service identity
  - provides permission lists, i.e. the access right for each service, given a user id
- Service
  - provides useful data and/or receives data and commands that can trigger processing
  - perform call-backs when it is expected a long processing time
- Local Controller
  - caches the permission list
  - checks whether requests are legit, calls a service, or returns cached results if available and within their validity time

- caches results of invoked services and holds the validity time for such results
- provides methods to be called-back when services provide results asynchronously

- Permission
  - holds the permission list related to some user, and its validity time
- Client
  - is the portion of the application running on a user device
  - allows the user to initiate several operations

Dynamics

Figure 3 shows the sequence diagram for the initial phase when user authentication and the caching of the permission list is needed. Accordingly, methods `login()`, `getAuthorisat()`, and `set()` are invoked. Figure 4 shows the subsequent requests, when permission checks are performed locally, being *LocalController* informed on the user permissions, and an asynchronous request to service `getData()` is performed, just after access has been granted by invoking method `check()` having as parameters user id and service id.

In case the needed service reply has already been cached within *LocalController*, a simpler sequence of invocations would be needed, i.e. *LocalController* will have to check permissions first, then look up

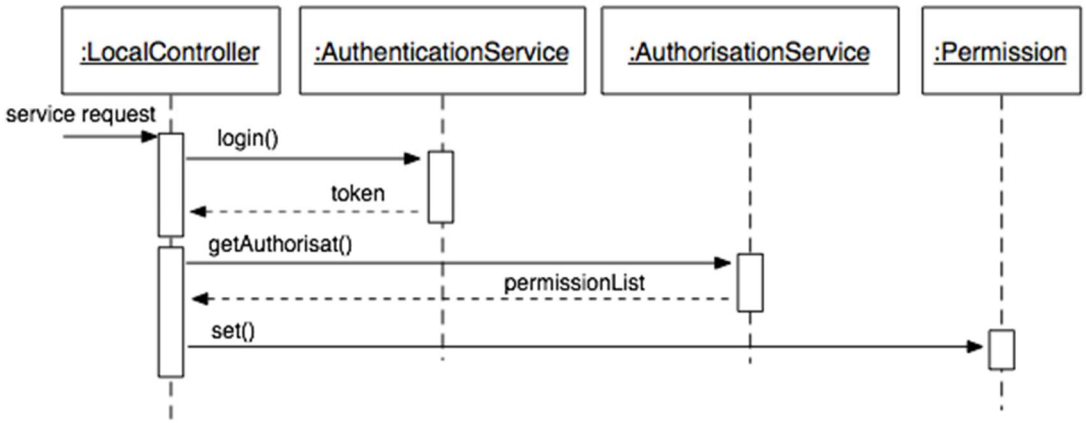

Figure 3. Sequence diagram showing authentication and access control list initial request

the reply and check its validity time.

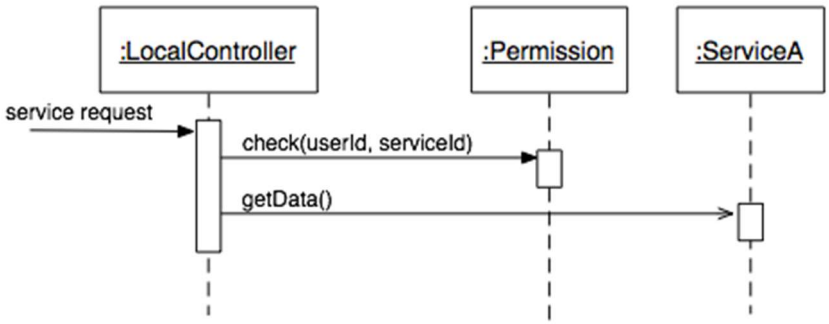

Figure 4. Sequence diagram showing local permission checks, after the initial request

Known uses

Data caching has been widely used for speeding up replies, when it is expected that otherwise computing time or communication delays can be costly [Tate]. The suggested solution brings a novel use of such an idea, since it considers caching the permission list, handled as data for access control. Asynchronous calls are suggested for possible unreliable, unresponsive services [Bloch, POSA2].

The proposed design pattern has been implemented as a solution on several web-based applications by a local company providing systems for e-commerce, libraries, etc.

Consequences

Pros

The effects of possible network congestions are reduced, since there are less messages sent to a remote and centralised authorisation service. The workload given to the authorisation service is lowered since for each logged user only one request is performed for the services located on one host.

The workload offered to the server host is lowered by the proper handling of cached results from services. According to the application at hand, useful results could be cached across several users and sessions to enhance the benefit of caching.

Service replicas are easier to implement thanks to the call-back mechanism adopted by asynchronous calls. I.e. the reference to the caller can be passed on to a newly created instance of the service on a different host.

Cons

LocalController has the burden to cache replies for several services, this task could be handled by a dedicated component.

A change to the permission list on the authorisation service will have to make invalid the local copy held by the *LocalController*. Therefore, additional communication is needed to let the authorisation service make *LocalController* list invalid, then a proper method should be implemented within *LocalController* to support it.

Related patterns

Authorisation pattern allows an application to ensure that only specific clients can access the functionality of a subsystem [SecPatt, p. 245] [POSA4, p. 351]. Reference Monitor [SecPatt, p. 256] and Reified Reference Monitor [SecPatPra] patterns describe the solution to enforce access restrictions.

Such components offer valuable support and have been integrated in our solution, which includes a local cache to avoid or minimise delays due to remote permission checks.

Capability-based access and the Capability pattern regulate access to a service by providing a client with a ticket for the service [SecPatPra]. Once the rights given to clients have to be checked for authenticity on the server-side, to avoid network-related delays and bottlenecks our proposed solution could be of benefit for such a check.

Resource Cache pattern lets us minimise the cost of repeated accesses to the same resource [POSA4, p. 505]

Asynchronous Completion Token pattern allows an application to efficiently process replies of asynchronous operations invoked on a server [POSA2, p. 218]

Proxy pattern can be used to cache data coming from the *Real Subject* [GoF, p. 207] [Roles].

3. Conclusions

This paper has described a solution for securing accesses to services while tackling issues that can degrade performances. Firstly, the access control list is cached, and this drastically decreases the number of requests to a central authorisation service, and network based communications. Additional support for performance is provided by caching the service replies, and associating them a validity time. Finally, it is suggested to perform asynchronous calls to computationally costly services.

The observations of the runtime behaviour of various occurrences of such a design pattern on real software systems have shown a drastic increase in performance when compared with a straightforward simpler implementation.

This design pattern aims at assisting the developer tackle security issues without compromising performances and shed lights on possible bottlenecks when distributing services across a network.

Acknowledgments

This work has been supported by PRIME project funded within PO FESR Sicilia 2007-2013 Framework and FIR project 375E90. The author thanks shepherd Shin-Jie Lee for his contribution during the preparation of this paper, and the writer's workshop participants Joe Yoder, Rebecca Wirfs-Brock, Yung-Pin Cheng, G. Priyalakshmi, Eduardo Fernandez-Buglioni, and Hironori Washizaki for their comments.

References

[Bloch] Joshua Bloch. Effective Java (the Java series). Addison-Wesley, 2008.

[GoF] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: elements of reusable object-oriented software. Pearson, 1994.

[POSA2] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2. John Wiley & Sons, 2000.

[POSA4] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing, Volume4. John Wiley & Sons, 2007.

[Roles] Rosario Giunta, Giuseppe Pappalardo, and Emiliano Tramontana. Aspects and Annotations for Controlling the Roles Application Classes Play for Design Patterns. In Proceedings of 18th IEEE Asia Pacific Software Engineering Conference (APSEC), 2011, pp. 306-314

[SecPatt] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. Security Patterns: Integrating Security and Systems Engineering. John Wiley & Sons, 2006.

[SecPatPra] Eduardo Fernandez-Buglioni. Security patterns in practice: Building secure architectures using software patterns, Wiley Series on Software Design Patterns, 2013.

[Spring] Rod Johnson, Juergen Hoeller, Alef Arendsen, and R. Thomas. Professional Java Development with the Spring Framework. John Wiley & Sons, 2009.

[Tate] Bruce Tate. Bitter Java. Manning Publications, 2002.

[YWW] Joseph W. Yoder, Rebecca Wirfs-Brock, and Hironori Washizaki. QA to AQ Part Five Being Agile at Quality "Growing quality awareness and expertise". Proceedings of Asian Conference on Pattern Languages of Programs (AsianPlop). February 24-26, 2016. Taipei, Taiwan.