A Near-Real-Time Behaviour Control Framework

Bastian Preindl
Institute of Software Technology
and Interactive Systems
Vienna University of Technology
Austria
preindl@ifs.tuwien.ac.at

Alexander Schatten
Institute of Software Technology
and Interactive Systems
Vienna University of Technology
Austria
schatten@ifs.tuwien.ac.at

Abstract

The NuBric behaviour control framework (BCF) is a near-real-time framework written entirely in Java based on the paradigms of both role based access control (RBAC) and policy based access control (PBAC) which has been designed to be open and extensible by third-party-modules. Its purpose is to protect resources of any kind by session-specific access restriction and behaviour control.

In short a user or process has to preliminarily connect to the framework and the framework decides whether the user or process is permitted to access the protected resource or not. If the permission is granted the user or process is able to directly access the resource but will be controlled and regulated during the access, so NuBric acts as behaviour control between user or process (session) and resource.

NuBric by design does not protect access to objects in memory nor is it deployable within another application framework. It constitutes a standalone near-real-time framework to restrict access to external, framework-independent resources by controlling and triggering external, framework-independent facilities.

1 Introduction

The purpose of the NuBric framework is to provide fine-grained access control to resources based on users' or processes' roles and their behaviour in the system. These resources may be hardware (computers, devices, communication facilities) or software (files, processes, threads).

Nearly every resource shipping today is equipped with its own access control or security features. The purpose of NuBric is not to substitute them, it moreover needs these features for its control ability. The NuBric framework models an internal object representing the accessing client based on properties like a UID, an IP-address, even a username. When gaining access to a NuBric-controlled resource the approaching user (or any other object like a process) is going to be examined and classified. Relying on the information which is fetchable about a user and how this information is constituted, the resource access abilities are set.

The NuBric framework has been designed and implemented to be flexible, versatile, scalable, portable, platform independent, easy to configure, setup and bring into operation and—as most important aspect—easy to extend.

To achieve these objectives the framework has been designed in an object-oriented way and implemented using Java. Java offers easy to handle ways like dynamic class loading and reflection to dynamically extend the framework even during runtime.

NuBric's philosophy is to act as *delegating* and *controlling* instance (access decision facility—ADF) for and between access requests (modelled as session object), protected resources and policy enforcing facilities. The framework and the framework's modules do not enforce policies theirselves like access enforcement facilities (AEF) do—they rather trigger external devices and facilities to enforce policies (see figure 1). "Outsourcing" the policy enforcement by near-real-time reconfiguration of external devices and facilities is leading to maximum transparency, performance and stability [10].

The NuBric resource access framework (RAF) is the completely re-engineered successor of the Irongate AAA Architecture published first time in 2004 [4]. Whereas the architecture described in [16] was mainly focused on network access control, the NuBric framework underlies no boundaries concerning it's field of application. The basic idea of NuBric has been enhanced since 2001 and implemented and tested within academic environments and from a scientific point of view since 2002.

NuBric's development state is alpha by November 2006. By now most parts of the framework architecture are implemented and the framework is runnable in stable way. Some parts are still open to changes for the next few months

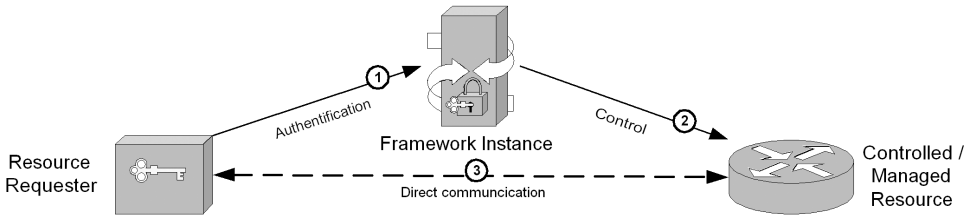

Figure 1. *NuBric's role within a secured environment: Before gaining direct access to a resource the access requester has to authenticate itself within the framework (1). After successful authentication the framework enables the requester to directly access the resource (3) by enforcing access policies (2)*

(method invocation sequence, module loading, configuration) or not implemented yet (clustering).

2 Behaviour Control Scenarios

In principle NuBric has been designed to be a framework working in near-real-time, controlled by it's watchdog, but is also able to be reduced to e.g. an authentication tool not supervising access to resources after giving permission. Hence, the framework is capable to handle every kind of access and behaviour control, whether it has to be done in near-real-time or not.

The reason why the framework is able to work and react in 'near-real-time' but not 'real-time' is the absence of guaranteed resources and reaction times which would be needed to satisfy the requirements of a real-time system. The framework's watchdog is querying the system's hardware-clock to schedule the method invocations which on their part act as soon as the required resources are present.

The framework's API is optimized to enforce the design of highly reusable modules. So modules written for one scenario may be reused in another, completely different, scenario. Also the delivery of module packages covering a whole topic is supported and internal code reuse is easily possible. E.g., when a module is developed to access a specific database to retrieve session specific data it can be reused in various access control scenarios like ssh login or web-login.

The following use cases are short, very different examples what NuBric is capable of if the right modules are implemented.

2.1 Detection of application-policy abuse

Service providers define policies for usage of their services (e.g, in web-mail services like Yahoo mail, users forbid the usage of robots to automatically create email accounts in their Anti-SPAM policy [19]). However efficient policy execution is a challenging task. The somewhat tricky thing about automated abuse is that the actions of one agent are not harmful and probably even within service policy boundaries. The same actions taken by an endless count of agents are indeed harmful and dangerous.

A typical kind of abuse of web-applications is the installation of intelligent software agents to multiply abuse activities which not seldom cause respectable harm to infrastructures and businesses. NuBric may provide countermeasures on two levels: First, access control is not only provided at login-time, but running sessions can be observed and interrupted by the framework. Secondly, complex behaviour-evaluation can be implemented, which is not restricted to observation of a singe user account.

Following the example above a module could be implemented that analyses the behaviour of multiple sessions: If abuse patterns are detected, the access control module could eventually terminate running sessions of clients involved and lock them out of their accounts plus inform administrators about the incriminating activities.

2.2 Network interconnection control

As NuBric's origin lies in network access control (NAC) and network access protection (NAP), the scenario of controlling the access to the Internet by focusing on the local gateway as main policy enforcement point (PEP) to control a network user's behaviour is standing to reason.

In this use case the framework's policy enforcement point(s) are the local Internet gateway, its packetfilter, traffic shaper, DNS-service and routing control in detail. The user authentication at the framework may be handled in several ways, e.g. by providing a module handling incoming connections initiated by a proprietary client software which passes username and password or by interacting with other applications like mail server and web-services (refer to the scenario "single-sign-on").

When an authentication-procedure is instantiated by the

user to gain access to a NuBric-managed Internet connection, a new user object is instantiated within the framework and fed with every information the client-interfacing module is able to collect from the client (user). Important values are IP-address, MAC-address, username and password for sure. The IP- and MAC-addresses are later used to set the access rules by dynamically reconfiguring the system's packetfilter (a similar approach is described in [11]).

After graceful authentication the policy rules are dynamically calculated depending on accounting information collected about the user and applied by several policy enforcement points (or policy enforcement tools) what enables the user to (probably restrictedly) access the Internet. During the user's authenticated session its (traffic) behaviour is continuously monitored and evaluated by polling values like packet and byte counters, firewall log facilities and connection lists. As long as the user's behaviour is within both dynamically calculated and preset policies (where these policies are being continuously refreshed) and the user remains online (doesn't autonomously quit the session) his ability to natively access the Internet is uninterrupted. If its behaviour (even slightly) exceeds the boundaries set, the session is interrupted by the framework and the policy enforcement points disable the user to continue its Internet access.

A precondition is for sure, that modules for all functions needed are already loaded by the framework. Modules we could need are backend-connectors for e.g. LDAP and mySQL, time and volume calculators, packetfilter and shaper controllers, byte counter evaluators, accounting modules a.s.o.

2.3 Enterprise framework integration

This scenario is at the moment some kind of experimental. The idea is that the NuBric framework is integrated in an enterprise application framework by communication using a standard protocol like JMS. A connector is deployed as Servlet which interacts with the NuBric framework (as communication protocol the SOAP-based security exchange protocol suggested by Suzuki et al. is intended [18]). Other servlets or beans authenticate whatever they want (or trigger whatever they want) by logging into the framework or registering in the NuBric framework using JMS.

One sophisticated way of enterprise framework integration would be the configuration of a web-application request interceptor like valves (Tomcat) or filters (Servlet 2.3 specification [2]) which enables a completely transparent access and control for every service-request. The idea is to intercept the invocation of the servlet before the servlet is called and evaluate the request in NuBric at that place. NuBric can now perform arbitrary analysis to decide whether the user is allowed to access this resource or not.

The interesting aspect of this approach is that the concerns are clearly separated, as the web-application is not aware of this access control mechanism. Additionally, if Tomcat valves are used (not filters), the configuration of this access control mechanism takes place in the server and not in the servlet configuration

2.4 Single-sign-on/off

Providing reusable modules for web services and application servers and e.g. a module which acts as or interacts with a daily secure authenticating procedure of the user like checking for new emails when arriving in the office or using a smartcard for entering the office could substitute all other authentication procedures usually needed to gain access to the needed resources for work.

When IMAP(S)-authentication is executed successfully the framework enforces all controlled resources like the company's web service, the local file server, the Internet access control to accept connections established by the user's machine which has been authenticated via IMAP. Also in this case modules which are already implemented to satisfy other use cases can easily be reused.

Regarding security requirements which have to be satisfied in the context of e.g. online banking the perspective of providing a low-level single-sign-off facility driven by NuBric and once more use-case-unspecific modules is standing to reason. If an online banking user's misdemeanour is not possible to be cut off by the web-application serving the user's requests caused by design flaws or unexpected software behaviours an emergency routine could be initialized by e.g. the web-application's control facility. This emergency routine interfaces with our framework and the framework on its part sets packet filters to cut off the misconducting user's session beneath the application layer.

3 Related work

NuBric integrates paradigms and principles which are well-defined in the past to define the term *(user/session) behaviour control*, which is novel in the field of security and access control.

By abstracting every access attempt or controlled session to a session object NuBric acts like a role-based access control (RBAC), as described in [3]. Nabhen and Jamhour furthermore combine RBACs with policy-based access control (PBAC) in [14] (calling this RBPIM: The role based policy information model) and therefor get closer to our framework's approach, but without mentioning the near-real-time component and behaviour control. By considering rules defined in connected data-sources in relation to identified session objects NuBric has some influences of rule-set based access control (RSBAC) like mentioned in [15] and [12].

Unlike [1] NuBric is not only based on events but moreover controlled by an autonomous management instance providing abilities to act in near-real-time.

The Securent Entitlement Manager [17] which has been developed by Securent in 2006 provides a quite similar approach focused on application access control based on the eXtensible Access Control Markup Language (XACML) [13], SOAP and the Security Assertion Markup Language (SAML) [8].

4 Architecture

The NuBric framework is implemented as multi-threaded native Java-application providing a container to deploy modules. The kind of the deployed modules define the frameworks application domain. Without modules the framework is able to run, but unable to act in any way, so normally at least a module to hot-plug new modules is loaded at the framework's startup.

Hot-pluggable modules provide the functions needed for the different states an access gaining user or session (object) can obtain. At the moment these states are yet hardcoded and are namely (in execution order) undefined, logging in, authenticating, accounting, activating, being online and logging out.

The framework implements a controlling instance (called the "watchdog") which acts as action trigger for the plugged in modules. It decides which action of which module is initiated at which time.

The framework's Core handles framework's startup and shutdown and the intercommunication with other NuBric instances (clustering) and with the host system. It controls the Watchdog-thread and provides methods for a communication between modules and session objects.

The Watchdog-thread is the "near-real-time"-component of the framework. The watchdog is instantiated at startup and continuously cycles through all current connections and connection attempts (represented and referred to as "session objects"). It inspects every session object concerning it's current state and decides if the state is going to be changed. At state-change the relevant external module's methods are invoked.

External modules are needed to interface and communicate with facilities outside the framework (e.g. policy enforcing agents/points, authentication backends, sessions or requests). When starting the framework without modules loaded during startup the framework can be considered "empty" and without practical functionality. An API is provided with the framework to implement third-party-modules which are deployable in the framework's module-container whether at startup or by hotplug during runtime. The purpose of loading modules is to define the application area and to add any functionality. Every module implements one or more of the following methods: (Session)Authentication, (Session)Accouting, (Session)Activation, (Session)Update, (Session)Deactivation. At least one module is implicitly needed to interact with the environment in any way.

The users or processes and their access attempts are abstracted as session objects within the framework and subject of all watchdog operations. Whenever a subject (session, process) wants to access the protected object (controlled resource) it has to preliminarily establish a connection to the framework instance protecting the object. This connection is mentioned as connection attempt and modelled as "session object" within the framework. When access permission is granted the subject usually establishes a direct connection to the protected object while passively or actively still staying in connection with the framework.

4.1 Module purposes

A module developed for the framework by using the provided API satisfies one or more of the following purposes as visualized in figure 4:

- *Authentication*—defined as authenticate(session)
- *Additional data retrieval*—defined as fetch(session)
- *Policy rule decision*—defined as account(session)
- *Policy enforcement*—defined as activate(session)
- *Near-real-time Behaviour control*— defined as update(session)
- *Logging*—defined as log(message)
- *Session connector or session interceptor*—by providing interfaces to initialize and keep alive controlled sessions
- *Framework management*—by providing external interfaces to interact with the framework's core.

An in some cases similar approach is proposed in [7].

Figure 3 illustrates the (time) correlation between the internal state of a session (object), the state-dependent method invocations and the abstract process flow of a session.

When a new session is instantiated whether actively or passively (also refer to figure 4 further on) a new session object is created, set to state *blank* (returned by *Core.newSession()*) and initial data is fetched by the module handling the session on client-side (**session connector or session interceptor**).

If all required parameters are present after a given time (signaled by the invocation of *Core.login()*) the state is changed to *loggingon* and *Modules.authenticate(Session)* is

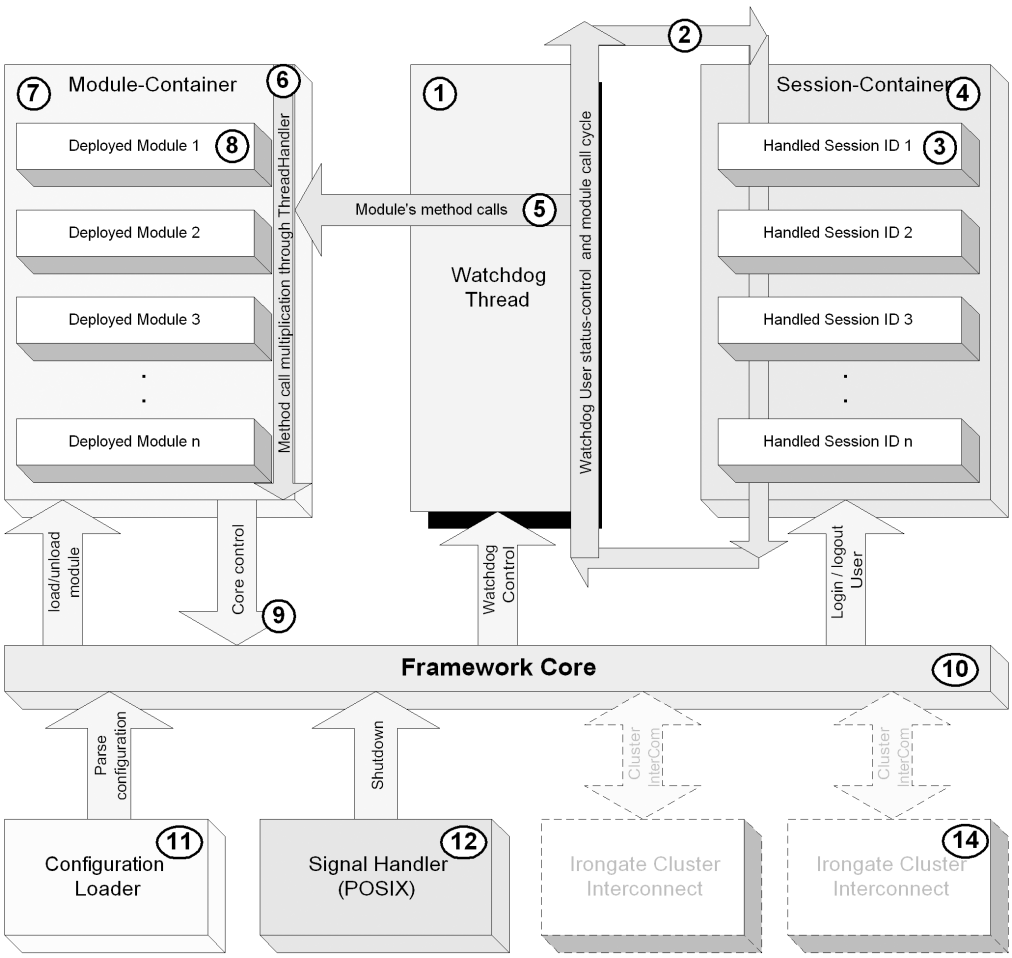

Figure 2. *NuBric's framework architecture: The WatchDog-thread (1) continuously cycles (2) through all currently active sessions (3) stored in the SessionContainer(4). If the current state of a session requires an action the WatchDog triggers (5) the ModuleContainer (7) to invoke the proper method (6) of every deployed module (8). If a module needs more information about a session or wants to perform framework management operations it communicates with the framework's core (10) via a specified interface (9). To deploy framework modules at startup the configuration file loader (11) is instantiated. To control the framework directly without using a framework management module a common POSIX signal handler is established (12) while framework interconnector (13) will facilitate clustering of multiple framework instances locally or remotely.*

invocated (**authentication**). Otherwise the session is going to be deactivated by invocating *Modules.logout(Session)* and the state is set to *offline*.

If the authentication was successful (signaled by a module's method calling *Core.authenticated(Session)*) after a given time (except for state *active* every state provides a preset timeout not mentioned further) *Modules.account(Session)* (**additional data retrieval** and **policy rule decision**) is invocated and the state is set to *accounting*.

This procedure is repeated for *activating* (**policy enforcement**). After shifting the session state to *active*, *Modules.update(Session)* (**behaviour control**) is invoked continuously while the session is active.

After session termination whether by a module (e.g. **framework management**, **policy rule decision**, **behaviour control**) or by the session itself (e.g. volitional logout, connection interruption) *Core.logout(Session)* is invoked (**policy enforcement**, **additional data retrieval**). The session state is changed to *deactivating* and after logout method invocation to *offline*.

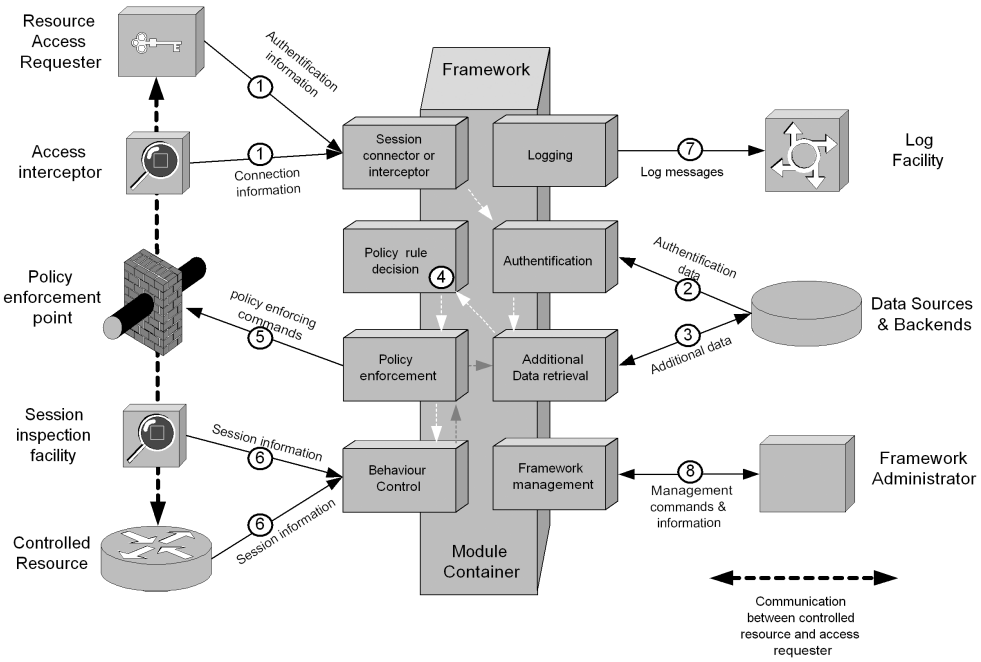

Figure 4. *NuBric's framework modules and their purposes: A new session is established whether actively by authenticating at the framework via a direct connection or passively by being tracked by a an access interceptor (1). In both cases a session connector or interceptor module is needed. Authentication modules authenticate (2) a session by querying external data sources (e.g. an LDAP backend) based on information collected and additional data is retrieved (3) by modules also accessing external sources. After successful authentication the policy rule decision takes place (4) in specific modules what is a precondition for triggering the policy enforcement points (5) in the next step. From now on a session is deemed to be established. During an established session behaviour control modules collect and evaluate information by querying session inspection facilities or the controlled resource itself (6). When a session is going to be terminated whether by framework's or session's decision the policy enforcement points are triggered to remove special session-specific rules (what immediately disables a direct connection between requester and resource) and a session summary is passed to external data sources (also known as accounting). During all framework, module and session operations logging may take place which uses external log facilities to distribute and communicate log messages (7). Framework management sessions (8) are established concurrently via special modules.*

After a predefined time the session object is removed and the session is finished.

4.2 The watchdog thread

WatchDog is started concurrently at the framework's initialization sequence and is established to control the state of every session object. In dependency on it's configuration and the session object's states it leaves an object untouched or changes it's state and invokes the properly method of all modules in relation to the session by calling ModuleTable.methodName(session). WatchDog cycles through all sessions and checks their states and the time they already remain in their current state periodically.

The origin of the NuBric framework is network access and near-real-time traffic control, so the currently hard-coded method-invocation-sequence (authenticate, fetch, account, activate, online, deactivate) is optimized for this kind of tasks.

WatchDog is also started and running even if no modules are loaded, but a running WatchDog without having any modules loaded is practically idle because there are no possibilities given to hotplug any modules during runtime without a loaded module supporting a module hotplug. So at least one module has to be loaded at startup to give any functionality to the framework.

4.3 Configuration

The configuration of the framework can be divided into *Core* and *Module* configuration.

The Core configuration is loaded and passed to the

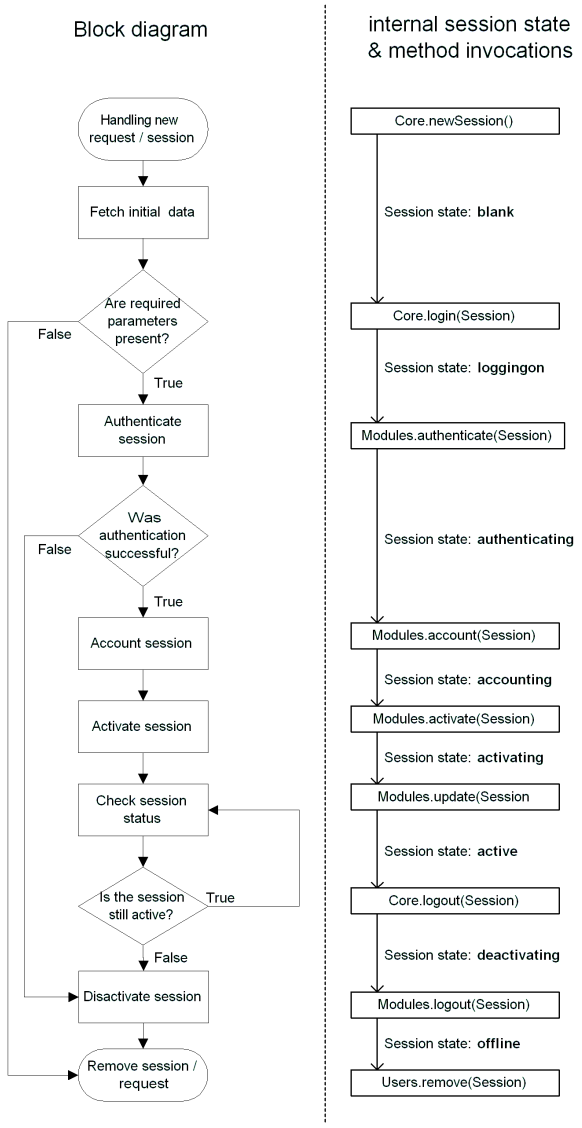

Figure 3. *NuBric's session handling sequence*

framework's core at startup and cannot be changed during runtime. It defines the behaviour of the framework concerning timings, states (at the moment only state timings) and other basic properties like the node's name. When loading a module the Core configuration (Properties) is merged with the module's specific configuration and passed to the module at initialisation.

The Module configuration is established by passing a module's configuration held in a Properties-container to the module at initialisation (by calling AModule.perpare() enforced by the Core). The configuration is whether set in the configuration file loaded at startup (if the module is loaded at the framework's initialisation) or via an extra module which is able to hot plug new module's. This module must pass a Properties-object when delivering the raw Class-object. This way of loading new module's during runtime is a base for configuration file independence. If e.g. the configuration file format is changed in future a proxy-module-loader simply has to be loaded to mix up different configuration formats.

4.4 Container classes

To manage the values coming across during the framework's runtime some individual container classes have been implemented. Values which have to be managed and controlled are *modules*, *sessions*, *session's parameters* and *timestamps*. So the following container classes are available:

- *ModuleContainer*—Every module loaded properly is referred to in the ModuleContainer. The ModuleContainer is needed by the WatchDog to call the proper methods of every module. Whenever a module is shut down it's going to be removed from the ModuleTable.
- *SessionContainer*—Every session managed by the framework is referred to in the SessionContainer. It's needed by the WatchDog to handle and manage all sessions on the one hand and by modules to directly access sessions by their session-ID on the other hand.
- *Session*—Every user connecting to the framework is represented by its own session object. This object contains every value concerning the session starting with its id and ending with every imaginable value of any possible datatype. Also a TimeStamp object is referred by the session object to tell the WatchDog when to change the object's states.
- *TimeStamp*—Acting as a kind of "marker" the TimeStamp tells the WatchDog when the last state-change has occurred for every session. So for every session exactly one TimeStamp is instantiated. The stamp may be read out and set.

4.5 Log facility

To avoid unwanted dependencies and interruptions caused by exceptions (errors, warnings ...) a log facility similar to log4j [5] has been designed. Every module may be part of the log facility as creating log messages on the one hand and processing log messages on the other hand.

Whenever anything happens a log message is created by calling Core.log(). Core.log instances a new LogMessage object and calls ModuleContainer.log(). The ModuleContainer then instances a new ThreadHandler for every module held in the container and the ThreadHandler calls the log() method of every module.

There is no exception thrown back to the calling method—every exception is caught and wrapped into a LogMessage-object. This is one of the framework's conclusions: It is more important that there is no framework operation interruption than having every error caught occurring in one of the modules. But the LogMessage doesn't only contain an exception occurred, it moreover holds several additional information about every message instanced.

By outsourcing the task of persistent logging to the response of third-party modules NuBric underlies practically no boundaries concerning the procession of log messages. This easily enables the integration of standards like log4j, syslog or any other established log facility.

The reason why not directly connecting to log4j but providing a proprietary log facility is the attempt of designing a slim and very independent framework with no requirement of external configuration files.

4.6 Multi-Threading model

NuBric has been implemented considering the need of high performance and stability. To comply with these requirements the framework heavily relies on threads. Whenever a method is invoked, a new Thread object is instanced only for processing this method. This design decision covers two important advantages:

1. The use of multiple threads enhances performance and makes the use of multi-core-processors possible

2. The whole framework doesn't depend on the performance and correctness of one single module. If one method-invocation crashes, whether the module nor the whole framework are affected in any way (surely when using synchronized blocks there has to be a mechanism to avoid dependencies).

To emphasise the advantages of thread-decoupling the following example is considered (refer to the sequence diagram displayed in figure 5):

Two different modules for authentication are deployed within the framework (e.g. one LDAP-connector, one RADIUS-connector). A newly established session is going to be authenticated and therefor the two modules' authentication methods are concurrently invocated. The first module confirms a successful authentication before exceeding the predefined maximum time but the second module's confirmation is delayed due to backend connection troubles. In a scenario where both thread executions are not decoupled nor running independently the session would whether be rejected due to a timeout or would have to wait until the second module has finished authentication even if the first module's authentication result was already satisfying. But in case of thread-decoupling the logon-sequence is immediately continued after successful authentication by the first module even if the second module hasn't finished authentication yet.

In figure 5 the procedures of method invocation and thread decoupling initiated by the watchdog-thread are illustrated in detail: Firstly the watchdog fetches the actual session list from the session container (whereas all session-concerned information is modelled internally as "user"), retrieves the first user object in the list and queries its current timestamp from the object itself. Whenever a timestamp has exceeded a preset limit its refreshed and the watchdog initializes the designated next method-invocation. It passes the current user object to the module container to instantiate one thread handler for every module's method to be invoked. The watchdog only waits until all thread handlers are instantiated and starts fetching and processing the next user object in the user list immediately. While the next user is already processed the thread handlers still treat the passed user object and method invocations independently until work is done and terminate afterwards.

The disadvantage of this design decision is the heavy amount of threads created, processed and removed continuously.

The *(practical) best case* is a thread-count of m (every method invocation terminates before the next session instance is processed by Watchdog), the *average case* is a thread-count of n*m (where n is the count of sessions actually active and m is the count of modules currently loaded) and the *worst case* is a thread-count of n*x (where x is the maximum thread-count per session defined in framework's startup configuration to avoid a system overload caused by a dead-locked or corrupt method invocation). The worst case can whether occur if the thread-cycle is configured much too short (so method invocations massively overlap) or if a method invocation can't terminate (cause of dependency problems e.g.).

$$\Omega(n, m, x) = m$$

$$O(n, m, x) = n * m$$

$$\Theta(n, m, x) = n * x$$

To optimise thread handling and method invocation performance a dedicated thread management is implemented in near future. The thread management will provide thread recycling, extended benchmark facilities and dead-lock detection.

5 Ongoing work

In near future the NuBric development level will reach beta state. Topics which have to be processed to achieve

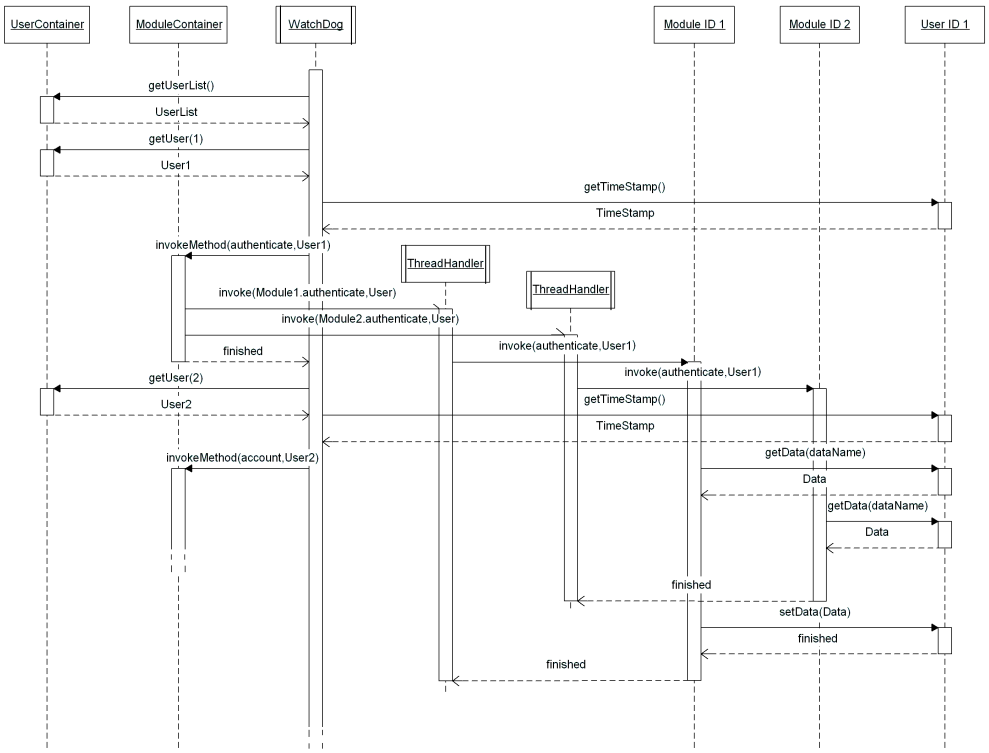

Figure 5. *Sequence diagram of one WatchDog cycle (example)*

this are: A final startup configuration syntax (probably using XML) including a dynamic configuration file creation engine, extensive benchmarks of the threading-model and dynamic method invocation, finalizing the API and making a decision concerning the dynamic class loading and method invocation based on the benchmarks achieved and find a proper model and algorithm for clustering and interconnection of multiple framework instances (probably using object spaces like GigaSpaces [6]).

Furthermore first-party-modules are going to be developed for NuBric to support JMX [9] for framework management purposes, SAML and XACML for authentication and log4j for logging purposes, to name only some of them.

6 Conclusion

The NuBric framework constitutes a novel approach in resource access and subject behaviour control.

Access control policies can be enforced applying a range of methods to evaluate the "behaviour" of the system or user that is trying to get access and allows to integrate a broad range of target systems that should be controlled. This allows not only to define access control policies at one central place, but also to use information (e.g., about behaviour) that comes from different connected systems to be used as an input for the access control logic. Moreover, in case of policy-violations NuBric can interact with the running session and stop access to all resources requested by the incriminated user and can eventually inform administrators to follow up the situation.

By combining other approaches like RBAC and PBAC in conjunction with the provision of third party module support via a comprehensive but lightweight API the field of application is almost endless. In contrast to many other projects and frameworks NuBric has no special focus of application and is therefor easily applicable for any environment by providing new, adopt existing or directly reuse external modules.

We have discussed several application scenarios in different application domains that show the flexibility of our system and how our approach provides advantages for all parties concerned (e.g. users, security consultants, developers). To emphasize the framework's capabilities and features its design and architecture have been described in detail where a special focus lied on different module's pur-

poses, container classes, configuration, logging, the watchdog and the extended aspects of thread decoupling.

References

[1] Manish Bhide, Sandeep Pandey, Ajay Gupta, and Mukesh K. Mohania. Dynamic access control framework based on events. In *ICDE*, pages 765–767, 2003.

[2] Danny Coward and Yutaka Yoshida. *Java Servlet Specification Version 2.4*. Sun Microsystems, Nov 2003.

[3] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[4] Reinhard Fleck, Norbert Jordan, and Bastian Preindl. Irongate AAA infrastructure. In *IEEE 59th Vehicular Technology Conference (VTC'04 - Spring)*, pages 1–5, 2004.

[5] Apache Foundation. The apache logging services. http://logging.apache.org/log4j.

[6] Giga-spaces object space. http://www.gigaspaces.com.

[7] Robert Grimm and Brian N. Bershad. Separating access control policy, enforcement and functionality in extensible systems. citeseer.ist.psu.edu/381709.html.

[8] Jeff Hodges, Prateek Mishra, Bob Morgan, Tim Moses, and Evan Prodromou. Oasis sstc: Saml security considerations.

[9] Java management extension (jmx). http://java.sun.com/javase/technologies /core/mntr-mgmt/javamanagement/.

[10] Angelos D. Keromytis and Jonathan M. Smith. Requirements for scalable access control and security management architectures. citeseer.ist.psu.edu/keromytis02requirements.html.

[11] Jahwan Koo and Seong-Jin Ahn. A network service access control framework based on network blocking algorithm. In *KES (3)*, pages 54–61, 2005.

[12] Leonard J. LaPadula. A rule-set approach to formal modeling of a trusted computer system. *Computing Systems*, 7(1):113–167, 1994.

[13] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah. First experiences using xacml for access control in distributed systems, 2003.

[14] Ricardo Nabhen, Edgar Jamhour, and Carlos Maziero. A policy based framework for access control. In *ICICS 2003*, pages 47–49, 2003.

[15] Amon Ott and Simone Fischer-Hbner. The rule set based access control (RSBAC) framework for linux.

[16] Bastian Preindl. AAA security framework for wireless access networks. Master's thesis, Institute of broadband communcation, Technical University of Vienna, Austria, August 2006.

[17] Securent. Securent entitlement manager. http://www.securent.com.

[18] T. Suzuki and R. Katz. An authorization control framework to enable service composition across domains. citeseer.ist.psu.edu/suzuki02authorization.html.

[19] Yahoo. Yahoo-mail anti-spam policy. http://docs.yahoo.com/info/guidelines/spam.html.