.NET integration TEAHA

Yoeng Woey Ho
y.w.ho@alumnus.utwente.nl

University of Twente
Dept. of Electrical Engineering, Mathematics and Computer Science
Pervasive Systems (PS)
PB 217, 7500 AE Enschede
The Netherlands

August, 2008

Abstract

The European Application Home Alliance (TEAHA) is a global project addressing networked home control applications, consumer electronics and A/V networked devices. It currently provides a central gateway that offers *Service Discovery (SD)* and interconnects different technologies in a secure way.

With the advent of Microsoft's .NET Framework an increasingly large number of devices are becoming .NET enabled. Devices ranging from portable Smartphone's, organizers, tablet PCs to full-blown computers are now including .NET technology. Microsoft's .NET offers programming language and OS independency, the latter due to the efforts of the open-source community. TEAHA's support for .NET technology must therefore not be neglected, as it is a valuable addition to the long list of supported technologies.

In order to compose a design that enables secure interaction between .NET and TEAHA devices, only communication protocols are used that are supported within both frameworks. After several protocols have been reviewed (including .NET Remoting, Web Services and Sockets), the UPnP SD protocol is considered to be the most suitable protocol; UPnP is supported on both frameworks through the UPnP OSGi bundle and Intel UPnP .NET stack.

Seeing that regular UPnP does not provide authentication or encryption, the concept of *Security Modules (SMs)*, and an authenticated version the Diffie-Hellman (DH) key exchange protocol were introduced to enable secure UPnP communication. The adapted SM provides several service hooks that are attached and integrated within the UPnP bundle and UPnP stack. Depending on the security settings the SM intercepts incoming and outgoing messages and applies cryptographic processing.

Moreover, if the SM is implemented in hardware, protection against cloning is offered while resource-restricted devices are relieved from intensive encryption and authentication processing.

The resulting design based on the UPnP SD protocol, SM concept and STS Key Exchange Protocol offers transparent and secure communication with policy enforcement between .NET and TEAHA devices and service. Furthermore, the design relies on open internet and non-propriety standards and software, and offers support for both action and event driven service interaction.

Seeing that policy enforcement, and conversion of service requests and responses is entirely handled by the central TEAHA gateway, the design is considered to be limited in terms of scalability. The increase of the number of connected TEAHA and UPnP devices will eventually require expanding the resource capabilities of the central TEAHA gateway or by distributing the service load amongst additional TEAHA gateways.

Samenvatting

The European Application Home Alliance (TEAHA) is een Europees project dat tracht om de onderlinge communicatie tussen netwerkapplicaties, audiovisuele netwerkapparaten en consumentenelektronica te bevorderen. TEAHA biedt een centrale gateway aan met als kern, ondersteuning voor *Service Discovery (SD)* en daarnaast de mogelijkheid om verschillende technologieën op een beveiligde manier met elkaar te laten communiceren.

Sinds de introductie van Microsofts .NET-technologie, heeft deze in korte tijd veel draagvlak en populariteit verkregen. Thans wordt .NET door vele apparaten ondersteund; reikend van Smartphone's, organizers, tablet pc's tot aan de huiscomputer.

Het .NET-raamwerk biedt een platform aan dat programmeertaal-onafhankelijk is en de communicatie tussen verscheidene apparaten met behulp van *Web Services (WSs)* en .NET Remoting aanzienlijk vereenvoudigt. Dankzij de snelle groei en opkomst van .NET, is de ondersteuning voor .NET-technologie inmiddels onmisbaar geworden in TEAHA.

Het ontwerp, welk beveiligde communicatie mogelijk maakt tussen .NET en TEAHA-apparaten, is enkel gebaseerd op communicatie protocollen die ondersteund worden door de beide eerdergenoemde raamwerken. Na diverse protocollen met elkaar te hebben vergeleken (waaronder .NET Remoting, Web Services en Sockets) is het UPnP protocol het meest geschikte communicatie protocol gebleken; UPnP wordt door beide raamwerken ondersteund middels de UPnP OSGi bundle en Intel UPnP .NET stack.

Gezien het feit dat UPnP echter geen ondersteuning biedt voor authenticatie en encryptie, is er gebruik gemaakt van het Security Module (SM) concept en een geauthenticeerde versie van het Diffie-Hellman sleutel-uitwisselings protocol (STS) om beveiligde UPnP communicatie mogelijk te maken.

Het toe- en aangepaste SM concept biedt enkele service hooks aan die geïntegreerd worden in de UPnP bundle en UPnP stack. De SM voert, aan de hand van de toegepaste Security Mode, de vereiste cryptografische transformaties uit op binnenkomende en uitgaande berichten. Voorts, indien de SM in hardware is geïmplementeerd, zal deze naast het aanbieden van beveiliging tegen cloning ook resource-restricted apparaten ontlasten van intensieve encryptie en authenticatie handelingen.

Het uiteindelijke ontwerp is gebaseerd op het UPnP SD protocol, SM concept en STS sleutel-uitwisselings protocol om transparante en beveiligde communicatie met policy enforcement tussen .NET en TEAHA-apparaten en services te faciliteren. Bovendien, maakt het ontwerp gebruik van open Internet en non-propriety standaarden en software, en worden action alsook event driven service interactie ondersteund.

Aangezien policy enforcement en de conversie van service requests en responses volledig wordt afgehandeld door de centrale TEAHA gateway, wordt het ontwerp gekenmerkt als minder schaalbaar. De uitbreiding van het aantal aangesloten UPnP en TEAHA-apparaten zal uiteindelijk een toename van de resource capabilities van de TEAHA gateway of een toename van het aantal TEAHA gateways vereisen.

Preface

This report reflects the results of my master's thesis during my Computer Science studies at the University of Twente. The research leading to this report was done under supervision of ir. J. Scholten.

My gratitude goes to Hans Scholten (University of Twente) for supervising, and providing me with useful knowledge and his support, ir. P.G. Jansen (University of Twente) and Mr. Antonio Kung (Trialog, Paris, France) for being members of the graduation committee. I would also like to thank my family, friends and all whom have given me advice and support.

Yoeng Woey Ho
Amsterdam, August 2008

Contents

1 Introduction

A quiet and relaxing evening at home, watching a recorded movie on TiVo that aired last night. Unfortunately, you receive an incoming call on your mobile phone. A small window fades in on the corner of your screen, showing the name and picture of the calling person. You choose to accept the call using the remote control, while the recording automatically pauses and the mobile phone streams a live audio and video feed to the TV. After finishing the call, the TV switches back to the recording and continues playback.

Not unimaginable in today's modern world, where technology has taken a ubiquitous role in daily life. The global expansion of the Internet has indicated the growing need for worldwide information distribution and connectivity. However, on a more smaller and local scale, interconnectivity between home appliances, consumer electronics and multimedia applications is also becoming more popular; allowing personal media and device services to be accessed and controlled from anywhere within the home environment.

The European Application Home Alliance (TEAHA) is a global project addressing networked home control applications, consumer electronics and A/V networked devices. TEAHA's objective is to develop an open, secure, interoperable, and seamless global home platform. TEAHA's approach is to define a suitable middleware platform that allows the seamless interworking of a wide variety of appliances found in a home environment. It currently provides a central gateway that offers *Service Discovery (SD)* and interconnects different technologies in a secure way.

With the advent of Microsoft's .NET Framework an increasingly large number of devices are becoming .NET enabled. Devices ranging from portable Smartphone's, organizers, tablet PCs to full-blown computers are now including .NET technology. .NET offers programming language and OS independency, the latter due to open-source community efforts. TEAHA's support for .NET technology must therefore not be neglected and is a valuable addition to the long list of supported technologies.

The thesis revolves around .NET and TEAHA devices being able to discover and access each other's services. SD is an important concept as it covers the automatic detection of devices and services on a computer Network.

The design and prototype implementation of a .NET bundle for TEAHA, which enables interoperability between TEAHA and .NET, will be discussed. After discussing the requirements and relevant technologies, some design concepts will be introduced. Based on these design concepts a final design is composed that enables secure discovery and access between .NET and TEAHA devices and services. Concluding, a prototype is created that will implement the concepts depicted in the final design.

1.1 Assignment

The project's main intention is to offer a solution for enabling secure communication between TEAHA [1] and .NET [2] devices. The development of a .NET bundle, which is a software component for the *Open Services Gateway initiative (OSGi)* framework [3], will provide this functionality.

The TEAHA project uses the Java OSGi framework as a base for its own framework. Consequently, to design the .NET bundle, communication essentially needs to be realized between .NET and Java. Communication methods that may be of help, such as sockets, *Web Services (WSs)*, and .NET Remoting [4] are discussed in the technology overview.

As secure communication plays an important role in a distributed environment, the .NET bundle will also need to provide support for authentication and encryption.

Additionally, for demonstration purposes, a client application utilizing the .NET bundle is implemented in *.NET Compact Framework (.NET CF)*; which is a specialized version of the .NET Framework intended for mobile devices.

1.2 Approach

The .NET bundle enables communication between .NET and other devices connected to a TEAHA gateway. To develop a .NET bundle for TEAHA, an in-depth knowledge about .NET Framework and OSGi is required. Research on .NET Framework and OSGi is necessary to collect information about supported network and communication technologies.

Furthermore, requirements and limitations for using certain protocols need to be examined, in order to provide a base for the design and implementation of a prototype.

Several related technologies will be discussed in the technology overview. By examining current related available technologies, one can avoid dealing with problems which have already been solved. Furthermore examining these technologies offers insight on how to approach the design and which technologies should be included. After introducing existing standards and techniques related to this project, some design concepts will be provided and discussed. Based on these design concepts and examining the consequences of a certain design, a final design can be composed.

1.3 Structure of the Thesis

This report starts with introducing a set of requirements (chapter 2), which acts as a guideline for this project. Although, the requirements are stated in the beginning of the report, they were determined after an extensive research on .NET, OSGi and related communication technologies as depicted in the technology overview (chapter 3).

Based on a few of these technologies, several design concepts could be composed (section 4.2). Further by taking the requirements into account, the most suitable design concept can be chosen as a solid base for the final design (chapter 5) and eventually be implemented (chapter 6).

Lastly, this report provides a final conclusion (chapter 7) and several recommendations on future work (chapter 8).

1.4 Definitions and abbreviations

This section provides a list of abbreviations that are frequently used throughout this report.

API	Application Programming Interface
ACL	Access Control List
AKC	Agreement with Key Confirmation
BCL	Base Class Library
BSD	Berkeley Software Distribution
CA	Certificate Authority
CAS	Code Access Security
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLS	Common Language Specification
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CTS	Common Type System
DCOM	Distributed Common Object Model
DH	Diffie-Hellman
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Server
DTD	Document Type Definition
GENA	General Event Notification Architecture
HTML	Hypertext Markup Language
HTTP	Hyper Text Transfer Protocol
HTTPMU	HTTP Multicast over UDP
HTTPU	HTTP Unicast over UDP
IDL	Interface Description Language

IP	Internet Protocol
JIT	Just-In-Time compilation
JVM	Java Virtual Machine
KHMAC	keyed-Hash Message Authentication Code
.NET CF	.NET Compact Framework
OASIS	Organization for the Advancement of Structured Information Standards
OS	Operating System
OSGi	Open Services Gateway initiative
PDU	Protocol Data Unit
PKI	Public Key Infrastructure
REST	Representational State Transfer
RF	Radio Frequency
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAML	Secure Assertion Markup Language
SD	Service Discovery
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SSDP	Simple Service Discovery Protocol
SSL	Secure Sockets Layer
SSO	Single Sign-On
STS	Station-to-Station
TCP	Transmission Control Protocol
TEAHA	The European Application Home Alliance
TLS	Transport Layer Security
TTP	Trusted Third Party
UDDI	Universal Description, Discovery and Integration
UDN	User Device Name
UDP	User Datagram Protocol
UPnP	Universal Plug and Play
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
USN	Unique Service Name
UUID	Universally Unique Identifier
XML	Extensible Markup Language
W3C	World Wide Web Consortium
WCF	Windows Communication Foundation
WS	Web Service
WS-Discovery	Web Services Dynamic Discovery
WSDL	Web Services Description Language
WSE	Web Services Enhancements
XACML	Extensible Access Control Markup Language

2 Requirements

This chapter discusses the requirement specification in order to structure and guide the project. The goal of this project is to come up with a feasible design for a .NET bundle for the THEAHA framework.

2.1 .NET Bundle Requirements

A summary of the requirements is given below, followed by a more detailed description.

1. The design must allow .NET devices to transparently access and discover TEAHA services and devices.
2. The design must allow TEAHA devices to transparently access and discover .NET services and devices.
3. Support for .NET devices and services must include .NET Compact Framework.
4. The design must allow enforcement of policies on service access and discovery.
5. The design must support action and event driven user-service interaction.
6. The design must not be based or use proprietary standards and software.
7. The design preferably uses protocols that are well supported by the .NET and THEAHA framework; natively or by means of external software components.
8. The design must be scalable in order to support expansion of the number of devices for simultaneously accessing, discovering and offering services.

Requirement 1: Discovery and access of TEAHA devices and services

The design must allow .NET devices to transparently discover and access TEAHA devices and services. This requirement states that .NET devices must be able to locate and access TEAHA devices and services, regardless of the underlying technologies that the devices use to offer their services. .NET devices may for example need to access Bluetooth or ZigBee TEAHA devices and services.

Requirement 2: Discovery and access of .NET devices and services

The design must allow TEAHA devices to transparently discover and access .NET devices and services. Similar to the first requirement this requirement states that .NET devices must be able to offer their services to TEAHA devices, regardless of the underlying technologies that TEAHA devices are using.

Requirement 3: .NET Compact Framework support

The design must include support for *.NET Compact Framework (.NET CF)* devices and services. The .NET CF is targeted for mobile devices and will be discussed in more detail in section 3.3. The framework is adapted to the resource constrains and special functionality of handheld devices. Mobile devices are an important device category, as they are very suitable for offering an all-round mobile interface between users and TEAHA services. For that reason it is important to include support for .NET CF on TEAHA.

Requirement 4: Policy enforcement

The design must allow the policy enforcement on service access and discovery. Delivering secure service discovery is one of the important goals of the TEAHA project. The project provides security using enforcement of *policies*, which are rules that state under what conditions a service can be discovered or accessed.

The requirement for policy enforcement leads to the following detailed requirements:

4.a Checking of user- and service identification, encryption and authentication settings must be supported.

4.b Regular, encrypted as well as authenticated communication must be supported.

4.c Communication involving .NET and TEAHA devices and services must use special channels that provide hooks for policy and security enforcement.

Requirement 5: User-service interaction

The design must support action and event driven user-service interaction.
Within the *Service Oriented Architecture (SOA)*-model, there are entities requesting and entities providing a service. Generally speaking the entity requesting a service is called *user* while the entity providing a service is often referred to as *server*. Services can be divided into two types of user-service interaction: *event based and action driven*.

Communication initiated by the occurrence of events in the service is called *event based*. A user normally subscribes to a service, to receive notifications when a particular service event occurs. Within the context of the example, provided in the introduction, the mobile phone provides an event based *incoming_call* service. When the device registers on the network, the central gateway subscribes to the *incoming_call* service. If an incoming call is received, the mobile phone notifies all *incoming_call* subscribers. This type of event based communication plays an important role on *Universal Plug and Play (UPnP)* networks.

Communication initiated by the user performing an action is called *action driven*. Within the context of the provided example a hidden central gateway provides an action driven *answer_incoming_call* service. Upon receiving the incoming call, the person presses the ok button on the TV-remote to answer the call. The TV-remote (*user*) will then send a request to the *answer_incoming_call* service, which will trigger the sequence of events. Action driven is the default type of communication within networks like ZigBee and on frameworks like OSGi.

5.a The design must support event based communication.

5.b The design must support action driven communication.

Requirement 6: No proprietary software

The design must not use or be based on proprietary standards or software in order to avoid the payment of royalties or licensing problems. Open software with licenses like the *GNU General Public License (GPL)*, *Berkeley Software Distribution (BSD)*, or Apache license is thus recommended.

Requirement 7: Supported protocols

Preferably the design uses protocols that are well supported by the .NET and THEAHA framework. This includes native support or by means of available existing software components. Current supported protocols may be well developed, suitable and sufficient to offer communication between .NET and TEAHA devices. It would not be wise to create a design using on-the-edge unsupported protocols, just for the sake of using a new technology, unless they offer some distinct benefits.

Requirement 8: Scalability

The design must be scalable in order to support expansion of the number of devices for simultaneously accessing, discovering and offering services. The used protocols must be lightweight enough to support mobile and resource-constrained devices. They should however not restrict complex devices, such as personal computers, in their functionality. The designed system should also be scalable enough to fully service both small embedded as well as large complex devices.

2.2 Project boundaries

Aside from determining what the requirements are, it is also important to state the boundaries of the project in order to limit the scope of this project.

1. Taking requirement 4 into account, the design will provide security related hooks that can be used for adding security. Security is an important aspect of the design, and communication should be authenticated and encrypted if necessary. Furthermore, these security hooks will also offer a clear separation between security and basic communication functionality. The actual security and policy enforcement implementation will however be outside the scope of this project.

2. The final product of this project is a proof of concept, to demonstrate the possibility of integrating .NET into TEAHA. Therefore it is not necessary to work out all the minor irrelevant details.

Technology Overview

3 Service Oriented Architecture

The concept of *Service Oriented Architecture (SOA)* is an important aspect of this project. TEAHA/OSGi is an excellent example of a SOA that revolves around providing secure dynamic services. Moreover, the .NET framework also offers a SOA approach by including full support for *Web Services (WSs)*.

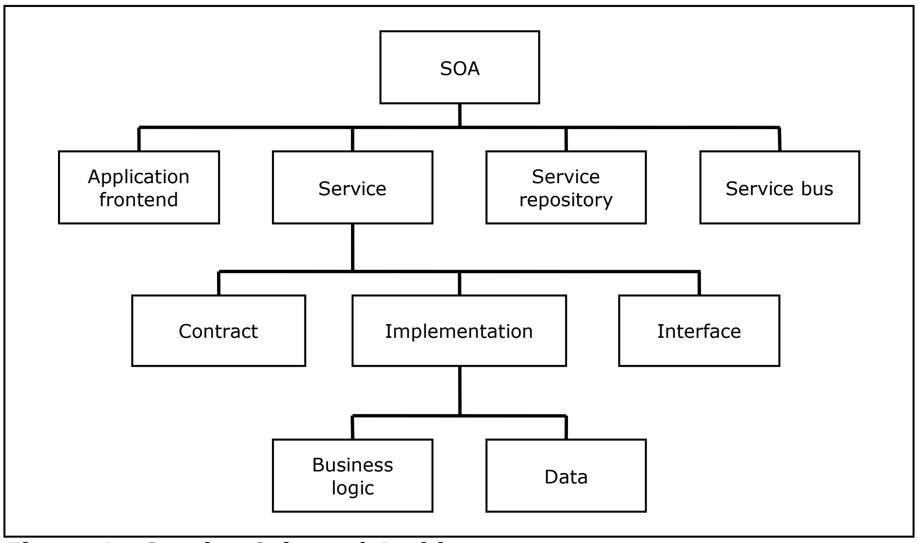


Figure 1: Service Oriented Architecture

SOA is a style of information systems architecture that enables creating applications that are built by combining interoperable and loosely coupled services. It supports integration and consolidation activities within complex enterprise systems, but it does not specify or provide a methodology or framework for defining capabilities or services.

This style of architecture promotes reuse at the macro service-level, rather than the conventional micro object-level, and simplifies interconnection and usage of existing IT legacy assets. The SOA is not tied to a specific technology and may be implemented using a wide range of technologies.

The key is independent services with defined interfaces that can be called to perform their tasks in a standard way, which maximizes the reuse of services. SOA separates the users from the service implementations, by using interface definitions that hide the implementation of the language-specific services. As a result, SOA-based systems are independent of development technologies and services can run on various distributed platforms and be accessed across networks.

The following guiding principles define the ground rules for development, maintenance, and usage of the SOA:

- Standards compliancy
- Reuse, granularity, modularity, composable, componentization and interoperability
- Services identification and categorization, provisioning and delivery, monitoring and tracking

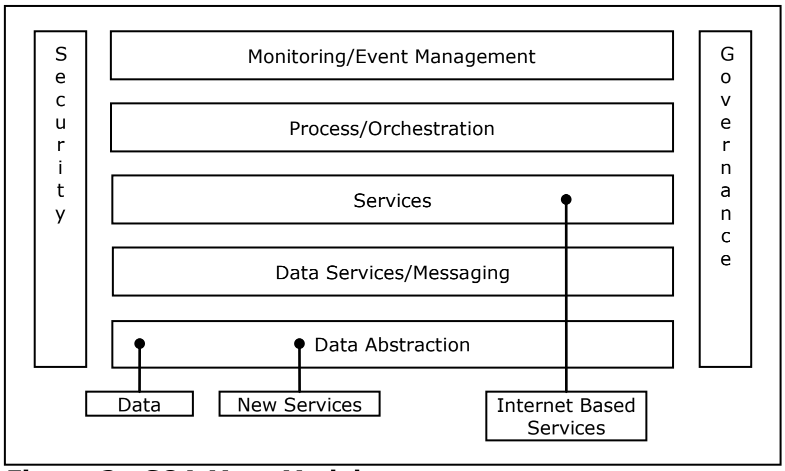


Figure 2: SOA Meta Model

In addition to the aforementioned guiding principles, SOA services should exhibit the following service principles [5]:

- Encapsulation
  Existing WSs can be consolidated and enclosed into SOA services, in order to be used under the SOA.
- Loose coupling
  Services maintain a relationship that minimizes dependencies and only an awareness of each other.
- Service Contract
  Services conform to a communications agreement, as defined by one or more service description documents.
- Abstraction
  Aside from what is described in the service contract, services hide logic from the outside world.
- Reusability
  Logic is divided into services with the intention of promoting reuse.
- Composability
  Form composite services by coordinating and assembling collections of services
- Autonomy
  Services have control over the logic they encapsulate.
- Optimization
  Services should be optimized as consumers generally consider high-quality services more preferable to low-quality ones.
- Discoverability
  Services are outwardly descriptive in order to be found and accessed via SD.

SOA and Web Service protocols
SOA is often defined as services exposed using the WS architecture. WS standards can be used to provide interoperability and some protection from lock-in to proprietary software, however, SOA is not limited to a specific technology and can be implemented using any service-based technology such as Corba, Jini and *Representational State Transfer (REST)*.

In sections 3.5-3.11 several communication methods including WSs are depicted, the latter are aside from .NET Remoting one of the main communication technologies for the .NET Framework.

Frameworks

This section introduces the two main important frameworks of this report. Seeing that TEAHA has been based on the OSGi framework specification, the first part will discuss OSGi and the dynamic services that are provided through software components. These software components are more commonly referred to as *bundles*.

Following, the .NET Framework, its mobile predecessor *.NET Compact Framework (.NET CF)* will be introduced; including support for mobile devices in TEAHA is regarded as a priority. As the .NET CF specifically targets the area of resource-restricted mobile devices, and primarily consists of a subset of the full .NET Framework, this project will consider the .NET CF as the framework for representing .NET.

Concluding, several alternative open-source Compact and full .NET Framework implementations will be introduced.

3.1 OSGi

This section describes the OSGi Framework and is based on the information provided by the *OSGi Service Platform - Technical Whitepaper* [6] and *OSGi Technology overview* [7].

The OSGi Alliance, formerly known as the *Open Services Gateway initiative (OSGi)*, is an open standards organization that has specified a remotely manageable Java-based service platform. The OSGi specifications define a standardized, component oriented, computing environment for networked services. The service platform offers life cycle management of software components in a networked device, from anywhere in the network.

A selection of ongoing OSGi work [6] that relates to this thesis:

- Web Services – Amazon, Google, Microsoft.NET, and many more WSs are becoming more popular each day. The OSGi Service Platform is an excellent platform for WSs. Its dynamic update facilities, the rich software environment that Java and the cooperative facilities that the OSGi Service Platform offers are an ideal combination with WSs.

- Connectivity – Embedded applications are ever more confronted with portable devices. IPods, mobile phones, PDAs are only a few examples of devices that people expect to collaborate with their car and home computers. Managing this complexity is one of the main goals of the OSGi Alliance.

3.1.1 Implementations

Currently there are a few open source implementations of the OSGi specification available: Oscar [8], Knopflerfish [9] and Equinox [10], the latter is also used in the popular Eclipse *Integrated Development Environment (IDE)* [11]. Currently Knopflerfish and Equinox are implementations of the OSGi R4 core framework specification.
TEAHA uses the Oscar implementation of the OSGi framework which is compliant with a large portion of the OSGi 3 specifications. Certain compliance work still needs to be completed [12]; although since the release of Oscar 1.05 in May 2005, further development seems to have been halted. Oscar also provides an incubator and repository for OSGi bundles that can be easily deployed into existing OSGi frameworks, called Oscar Bundle Repository [13].

3.1.2 Framework

The core of the specifications is the OSGi framework that defines an application life cycle model and a service registry. In addition, the framework can be extended with software components, which are called bundles. These bundles can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot.

A large number of OSGi services have already been specified: HTTP servers, configuration, logging, security, user administration, XML and many more. Standard OSGi framework-related service implementations are also provided, such as *Package Admin* and *Start Level*, but other non-framework related services are only available separately.

The OSGi Framework is the core component of the OSGi Specifications and provides a standardized environment for applications. As OSGi is intended for embedded device applications, the OSGi framework is small and reasonable lightweight, which makes it well suited for small devices.

The framework is also ideally suited for component- and service-oriented computing and can be easily embedded into other projects and used as a plug-in or extension mechanism; it serves this purpose much better than other systems that are used for similar purposes, such as *Java Management Extensions* [14].

The Framework is divided in a number of layers with an additional security system that is deeply intertwined:

- Execution Environment
- Modules
- Life Cycle Management
- Service Registry

Execution environment

Java Virtual Machine (JVM) provides the basis for the OSGi specifications, as it was the most logical choice when the OSGi Alliance was founded. The Java environment offers all the required features for a secure, open, reliable, well supported and portable computing environment.

Currently a possible candidate might be Microsoft .NET because it can provide similar features. However Microsoft .NET is mainly only available from one source, while a widely accepted open standard, for as many environments as the OSGi Alliance is targeting, requires an open multi-vendor platform such as Java.

Modules

The *OSGi Modules layer* is fully integrated with the security architecture and defines the class loading policies and adds private classes for a module as well as controlled linking between modules. The OSGi Framework is a powerful and rigidly specified class loading model and is based on top of Java but adds modularization. In Java, there is normally a single *classpath* that contains all the classes and resources.

OSGi bundles are dynamic software components that provide additional services. The main contents of bundles are the class files, which are the executable part of a bundle. In Java, classes are grouped in packages that have unique names.

Bundles can export and import packages; exporting packages makes them available to other bundles, therefore in order to import packages they need to be exported by other bundles. In case of multiple bundles exporting different versions of the same package, the framework selects an appropriate version for each bundle importing that package.

A package is always exported with a unique version; the importer can specify a range of compatible versions. The framework tries to minimize the number of exports, but it supports multiple class spaces where multiple versions of the same class can be in use at the same time. It also thoroughly verifies that a bundle cannot inadvertently get class cast exceptions to prevent clashes.

If a bundle exports a Java package and is subsequently uninstalled, then the framework ensures that importers are restarted so they can bind to a new package exporter. This whole process is transparent to the bundles because it happens when they are stopped.

Life Cycle

This layer adds bundles that can be dynamically installed, started and stopped, updated and uninstalled. Bundles rely on the module layer for class loading but add an API to manage the modules in run time. Extensive dependency mechanisms are used to assure the correct operation of the environment.

Installing (and uninstalling) bundles in a remote JVM provides the basis for networked services. Installing a bundle has two important aspects:

- The file format of a bundle
- Access to the install function

A bundle is typically stored in a Java Archive (JAR) file. Every JAR contains a Manifest, which stores information about the contents of the JAR in headers. The OSGi Alliance has defined a number of additional Manifest headers to allow the JAR file to be used in an OSGi Service Platform. Some headers are pre-defined by the JAR Manifest specification but the total set of headers is extendable and the values can be localized.

Access to the bundle installation function is done with an API that is available to every bundle. The Initial Provisioning specification or command line parameters can be used to install the first bundle and start an OSGi Framework implementation.

The API of the OSGi Framework is defined in the *BundleContext* object. The framework supplies this context object to the bundle when it is started. The context object has a number of methods to install new and list existing bundles.

The *installBundle* method takes a *Uniform Resource Locator (URL)* or *InputStream* as a parameter. The OSGi Framework inspects the headers and the code and installs the code in the OSGi Service Platform. After the bundle is installed, a Bundle object is returned. Once a bundle is installed, it can be started, but before classes in the bundle can be executed, the bundle must be resolved.

Service registry

The *Service Registry* enables the OSGi Service Platform to support applications built on the principles of SOA and allows objects to be shared between bundles. It allows application programmers to develop small and loosely coupled components, which can adapt to the changing environment in real time. The platform operator uses these small components to compose larger systems, while the *Service Registry* binds these components seamlessly together; it dynamically links bundles together while logging their state and dependencies.

Active bundles can use all standard Java mechanisms to implement their functionality. A bundle can become active at any time and provide a function that could be useful for other parts in the system. This makes the OSGi Service Platform, due to its dynamic nature, very suitable for many dynamic scenarios found in home automation, mobile and vehicle environments. A mobile phone might have an accessory plugged in, a residential gateway could detect a new media server or a car could detect the presence of a Bluetooth mobile phone that comes into range.

With aid of the *Service Registry*, bundles can:

- Register objects with the *Service Registry.*
- Search the *Service Registry* for matching objects.
- Receive notifications when services become registered or unregistered.

Objects registered with the *Service Registry* are called *services*. Services are always registered with an interface name and a set of properties. The interface name represents the intended usage of the service. The properties describe the service to its audience.

The OSGi Log Service would be registered with the *org.OSGi.service.log.LogService* interface name and provide properties such as *vendor=acme*, for example.
Discovering services is done with notifications, or by actively searching for services with specific properties. A filter language is used to select exactly the services that are needed. The *Service Tracker* utility class makes it easy to write code for this dynamic environment.

Registrations are dynamic and are dependent on the execution state of the bundles. The OSGi Framework automatically unregisters all services from a stopped bundle, notifying all its dependents.

3.1.3 Security

One of the goals of the OSGi Service Platform is to run applications from a variety of sources under strict control of a management system. A broad security model, present in all parts of the system, is necessary for running components in a shielded environment.

The OSGi specifications use the following security mechanisms:

- Java 2 Code Security
- Minimized bundle content exposure
- Managed communication links between bundles

Permissions

Security is based on Java and the *Java 2 Code Security* model. The language limits many possible constructs by design, like buffer overflows. In addition, it provides the concept of *permissions* that protect resources from specific actions. Permission classes take the name of the resource and a number of actions as parameters. Each bundle has a set of permissions that can be changed on the fly. Changing existing or setting new permissions is immediately effective. Permission assignments can also be done prior to, or just in time during the install phase.

When a class wants to protect a resource, it asks the Java Security Manager to check if there is permission to perform an action on that resource. The Security Manager then ensures that all callers on the stack have the required permissions. Checking the callers on the stack protects the resource from attackers. If for example, A calls B, and B accesses a protected resource, both A and B will need to have access to the resource.

The access modifiers in Java classes are used to protect access to code. Classes, methods, and fields can be made public, private, package private (accessible only by classes in the same package) or protected or (accessible only by sub-classes). The OSGi Service Platform adds an extra level of module privacy by making packages only visible within the bundle. These packages are freely accessible by other classes inside the bundle, but hidden from other bundles.

Package-sharing between bundles raises possible security vulnerability for malicious bundles. The OSGi specifications therefore contain *Package Permissions* to limit exports and imports to trusted bundles. It is a fine-grained permission that allows importing or exporting for a specific package, or for all packages.

Another security mechanism is *Service Permission* that allows bundles to register or get a service from the *Service Registry*. The permission is extensively used to ensure that only the appropriate bundles can provide or use certain services.

3.1.4 Bundles

Bundles are libraries or Java applications, packaged in a standard *Java Archive (JAR)* file, that can dynamically discover and use other components through the use of the service registry. Furthermore bundles can be remotely installed, started, stopped, updated, or removed on-the-fly without disrupting the operation or rebooting the device.

The OSGi Alliance has developed many standard component interfaces that are available from common functions like HTTP servers, configuration, logging, security, user administration, XML, and many more.

3.1.5 Services

The OSGi Alliance has defined many services that are specified by Java interfaces, which are implemented by bundles. Bundles register services with the Service Registry to allow clients to find these services, or react to them, when services appear or disappear. Each service is defined abstractly and is independently implemented by different vendors.

3.2 .NET Framework

This section is based on information provided by [15], [16] and [17].

Microsoft’s .NET Framework is a new software platform which largely unifies the development of web and client applications. It is a common environment for building, deploying, and running WSs and Web Applications; easing development of computer applications and reducing the vulnerability to security threats. Systems ranging from servers, Smartphone’s to wireless palmtops can use different versions of the .NET Framework in order to transparently interact with each other.

The following list shows the benefits and goals of .NET:

- Shared code and increased efficiency
- Robust code
- Secure execution
- Automatic deployment
- Rapid application development that requires fast time-to-market
- Ability to call Win32 DLLs without having to rewrite them
- Debugging and development can be done by Visual Studio.NET
- Code is not prone to fail due to uninitialized variables
- JIT compilation is not interpreted and runs code as processor native code
- Garbage collection greatly minimizes memory leaks by cleaning up inactive objects
- Support for encryption

3.2.1 .NET Architecture

The framework offers a layered, extensible and managed implementation of Internet services that can be quickly and easily integrated into applications. It provides a large body of pre-coded solutions to cover a large range of programming needs in areas including: user interface, data access, database connectivity, cryptography, numeric algorithms, network communications and web application development. Common class libraries, like *ADO.NET*, *ASP.NET* and *Windows Forms* are provided for including advanced standard services into a variety of computer systems.

Microsoft .NET Framework was designed with several intentions:

- Simplified Deployment
  Installation of computer software must be carefully managed to ensure that it does not interfere with previously installed software and that it conforms to security requirements. The .NET framework includes design features and tools that help address these requirements.

- Interoperability
  The .NET Framework provides means to access functionality that is implemented in programs that execute outside the .NET environment. Access to COM components is provided in the *System.EnterpriseServices* namespace of the framework, and access to other functionality is provided using the *Platform Invoke (P/Invoke)* feature.

- Common Runtime Engine
  Programming languages on .NET Framework compile into the intermediate language CIL, which is not interpreted but compiled into native code, in a manner known as *Just-In-Time compilation (JIT)*. The combination of these concepts is called the *Common Language Infrastructure (CLI)*; Microsoft's CLI implementation is known as the *Common Language Runtime (CLR)*.

- Language Independence
  The *Common Type System (CTS)* specification defines all possible data types and programming constructs supported by the CLR and how they are allowed to interact with each other. Because of this feature, the .NET Framework supports development in multiple programming languages.

- Base Class Library
  Base Class Library (BCL) is a library of types available to all languages using the .NET Framework. It provides classes which encapsulate a number of common functions, including file reading and writing, graphic rendering, database interaction and XML document manipulation.

- Security
  .NET allows for code to be run with different trust levels without the use of a separate *sandbox*, which is a security mechanism for safely running programs.

By isolating .NET applications (*assemblies*) from the *Operating System (OS)*, applications can be made processor and OS independent, and to be written in several programming languages while allowing seamless cooperation. First the source code is compiled to an executable that contains *Common Intermediate Language (CIL)*; formerly also known as *Microsoft Intermediate Language (MSIL)*. During execution, CIL is translated to native code for the particular processor and linked with the appropriate OS libraries.

Common Language Infrastructure

The most important component of the .NET Framework lies in the *Common Language Infrastructure (CLI)*. The purpose of the CLI is to provide a language independent platform for application development and execution; including components for exception handling, garbage collection, security and interoperability. Microsoft's implementation of the CLI is called the *Common Language Runtime (CLR)*.

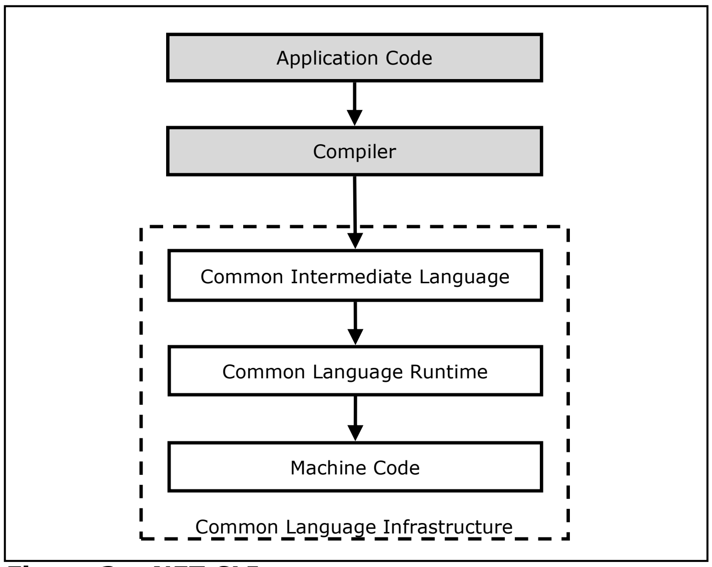


Figure 3: .NET CLI

Common Language Runtime

.NET applications execute in a software environment that manages the program's runtime requirements. The *CLR* provides the appearance of an application virtual machine and also other important services such as security mechanisms, memory management and exception handling. The CLR and class library together compose the .NET Framework.

- *Common Type System (CTS)*
  Set of types and operations, shared by all CTS-compliant programming languages.

- *Common Language Specification (CLS)*
  Set of base rules to which any language targeting the CLI should conform in order to interoperate with other CLS-compliant languages.

- *Just-In-Time Compiler (JIT)*
  Compiler which compiles code into native code during execution.
- *Virtual Execution System (VES)*

The VES loads and executes CLI-compatible programs, using the metadata to combine separately generated pieces of code at runtime.

3.2.2 Assemblies

The intermediate CIL code is housed in .NET *assemblies*, which are the .NET unit of deployment, versioning and security; for the Windows implementation this is analogous to *Portable Executabl*e files (*EXE* or *DLL*). An assembly consists of one or more files, and one of these must contain the *manifest* that carries the metadata.

The complete name of an assembly exists of the simple text name, version number, culture and public key token; the text name is required while the others are optional.

The public key token is generated when the assembly is created and uniquely represents the name and contents of the assembly file; therefore two assemblies with the same public key token are guaranteed to be identical. In the event of an assembly being tampered with, the public key can be used to detect the tampering.

3.2.3 Metadata

All CIL is self-describing through .NET metadata. The CLR checks on metadata to ensure that the correct method is called. Metadata contain all the information about assemblies and is usually generated by language compilers, but developers can also create their own metadata through custom attributes.

3.2.4 Namespaces and Class Library

The .NET Framework consists of a set of class library assemblies that contains hundreds of types and provide access to system functionality. The purpose of these class libraries is to provide a hierarchical namespace structure. These classes are language independent in order to accomplish cross-language inheritance and debugging.

System is the root namespace for the set of functionality that is part of this CLR platform. The *System* namespace contains the base *Object* that all others are derived from. The namespace also includes types for exception handling, garbage collection, tool types, console I/O, format data types, random number generators and mathematical functions.

3.2.5 Security

.NET has its own security mechanism, with two general features: *validation and verification* and *Code Access Security (CAS)*.

When an assembly is loaded, the CLR performs validation and verification. During validation the CLR checks that the assembly contains valid metadata and CIL, it also checks that the internal tables are correct. The verification mechanism checks to see if the code does anything that is unsafe. Unsafe code will only be executed if the assembly has the *skip verification* permission, which generally means code that is installed locally.

CAS is based on evidence that is associated with a specific assembly. Typically the evidence is the source of the assembly (whether it is installed locally or downloaded from the Intranet/Internet). CAS uses evidence to determine the permissions granted to the code. Other code can demand that calling code is granted a specified permission. The demand causes the CLR to perform a call stack walk: every assembly of each method in the call stack is checked for the required permission and if any of them does not have the proper permission then a security exception is thrown.

3.2.6 Windows Forms

.NET Framework enables rich client-side applications using Windows Forms, which is a subset of Windows Forms class library. By using Windows Forms, the underlying functionality of *Windows User* and *Graphics Device Interface* can be accessed.

Furthermore, elements of the *Graphical User Interface* can be dragged and dropped within Visual Studio.NET.

3.3 .NET Compact Framework

The Microsoft *.NET Compact Framework (.NET CF)* is the smart device development framework for Microsoft .NET, bringing managed code and XML WSs to mobile devices.

The framework is a subset of the .NET Framework class library and also contains features and classes specific to mobile and embedded development; as a result it implements approximately only thirty percent of the full framework [16].

As most portable handheld devices have limited processing, memory and storage capacity, the .NET CF has been specially adapted to the capabilities of these resource-restricted devices. The framework is optimized for battery-powered systems and avoids heavy use of RAM and CPU cycles. Binary sizes have also been largely reduced as the .NET Framework binary size is about 30MB, whereas the .NET CF is about 1.5MB.

3.3.1 .NET CF Architecture

.NET CF consists of the BCLs and a few additional libraries that are specific to mobility and device development, and runs on a high performance JIT Compiler. The CLR is built to be specific to the .NET CF for running more efficiently on small targeted devices that are limited in memory, resources, and must conserve battery power.

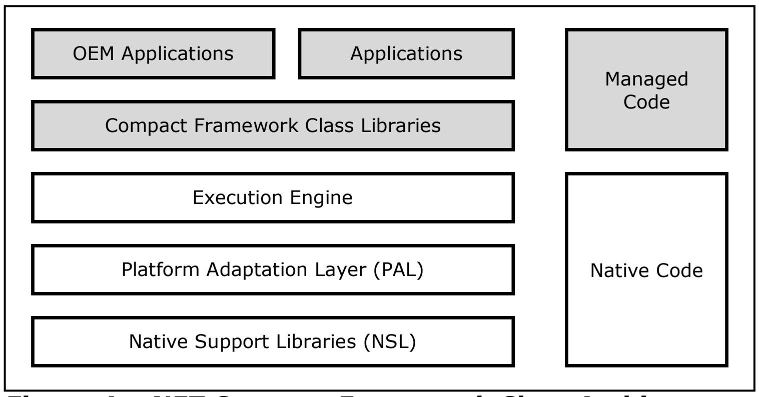

OEM Applications
Applications
Managed Code
Compact Framework Class Libraries
Execution Engine
Platform Adaptation Layer (PAL)
Native Code
Native Support Libraries (NSL)

Figure 4: .NET Compact Framework Class Architecture

The execution engine runs on top of the *Platform Adaptation Layer (PAL)* and *Native Support Libraries (NSL)*. The PAL contains a variety of subsystems that expose functionality of the underlying OS and hardware to the execution engine using an API set.

This allows the Compact Framework to be easily ported to various hardware platforms by vendors who provide platform-specific PALs. The NSLs are lower-level services which provide features that the execution engine requires but that might not be available on the OSs [18].

Besides the benefits and goals listed in section 3.2, .NET CF additionally provides:

- A portable subset of .NET Framework
- Same naming conventions as in .NET Framework.
- *Simple Object Access Protocol (SOAP)* support.
- Binary deployment runs on various CPUs on same platform without recompilation.
- Because recompilation is not necessary across .NET CF, controls, applications, and services can be easily moved from one device to another.

3.3.2 .NET Comparison

Both the .NET CF and the .NET Framework provide a consistent and familiar programming model. There are key similarities such as namespaces, classes, method names, properties, and data types. Programming is simplified and more secure due to the architecture and design of these platforms.

However, as the .NET CF targets handheld devices, one has to considerate the inherent memory, processing and storage constraints during design and implementation. Some guidelines on efficient programming and performance optimization, for the .NET CF environment, are available at [19].

Support for several important key features is missing in .NET CF:

- *Advance Graphics Device Interface (GDI+)*
- ASP.Net
- Asynchronous Delegates
- Multimodule assemblies
- Printing functionalities
- .NET Remoting
- Hosting WSs
- XML Schema validations and *XML Path Language (XPath)*
- No support for *System.Reflection.Emit* namespace.
- No interoperability with COM objects; however the *P/Invoke* method can be used to call native DLLs which in turn can access COM DLLs.

Communication

Communication within .NET Framework can be established by using WSs, .NET Remoting, *Windows Communication Foundation (WCF)*, or sockets. However, there are some important differences between the two framework versions regarding communication. On the .NET CF there is no support for .NET Remoting and raw sockets [20] [21], while WS support is limited to only WS access.

WSs, .NET Remoting, WCF, and sockets will be discussed in sections 3.8-3.11.
Moreover, in [22] a list of issues regarding sockets programming on .NET CF is given:

- Not all socket options are supported on all device OS's.

- The .NET CF is designed to be able to be ported to any number of OS's, each with their own levels of functionality. Therefore, the .NET CF does not limit the availability of socket options based on any particular level of support of an OS.

- Raw sockets are not supported.

- In .NET CF applications, the following options are supported but do not work without *Transmission Control Protocol (TCP) / Internet Protocol (IP)* stack modification and are reserved for future use: *AcceptConnection*, *ReceiveLowWater*, *ReceiveTimeout*, *SendLowWater*, *SendTimeout*, and *Type* [22].

Namespace

Some key features of .NET CF namespaces relating to section 3.2.4:

- The .NET and .NET CF use the same naming conventions for namespaces.

- Creates a logical, consistent namespace hierarchy that makes it easier for developers targeting the runtime to find and use.

- The same namespace can exist in different DLLs or assemblies and can contain multiple namespaces.

- Long namespaces have little impact on metadata bloat because the common prefix can be folded so that it is stored only once.

Windows Forms

.NET enables rich client-side applications using Windows Forms, which is a subset of Windows Forms class library. The .NET CF also supports Window Forms:

- All common features of Windows Forms in .NET Framework are present in .NET CF. As the framework targets mobile devices, these features are differently implemented to make them more efficient for size and performance.

- Windows Forms applications will not run managed forms that host native ActiveX controls as in the .NET Framework.

- Support for processor intensive features in .NET Framework is not present in the .NET CF to optimize size and performance; including GDI+ features.

Networking

The *System.Net.Sockets* namespace is used to provide an interface for accessing the transport layer of protocol stacks. To simplify common developer tasks, .NET CF provides additional classes that encapsulate much of the necessary code that is common across TCP client/server applications:

- TCPListener
- TCPClient
- UDPClient

The *HttpWebRequest* and *HttpWebResponse* classes offer a rich HTTP client, and support many standard encryption and authentication mechanisms, such as *Secure Sockets Layer (SSL) / Transport Layer Security (TLS)* and basic HTTP authentication. For implementing other Web requests the following interfaces can be used:

- WebRequest interface
- WebResponse interface
- IwebRequestCreate interface

These networking classes provide a synchronous and asynchronous development model, which simplifies the development model for programming by leveraging threads.

Threading

Threading is implemented in the *System.Threading* namespace, including support for creating and destroying threads. To find the current thread, the property of the *CurrentThread* on *System.Threading.Thread* is used. *System.Threading.ReaderWriterLock* provides synchronized access to data; *System.Threading.ManualResetEvent* provides control of flows from outside. Additionally, *GetHashCode* can be used for identifying a thread.

Applications can be started using the *System.Threading.Thread* class, while *System.Threading.ThreadPool* can be used for service applications that use threading to handle multiple client connections or client jobs simultaneously. *JoinMethod* is used to notify when a thread has terminated, however it is not supported in .NET CF.

Native Code InterOp

As most code is not written in managed code, there are occasions when code written in C or C++ (Win32 DLL or COM object) needs to be called. When .NET CF does not provide the required functionality, direct access to the OS or existing DLLs may be required.

The .NET CF supports a subset of the .NET Framework for direct OS and DLL access:

- *P/Invoke* is used to call Windows DLLs. *P/Invoke* functions and attributes are found in the *System.Runtime.InteropServices* namespace.

- *COM Interoperability* offers interaction with COM objects. However, because this feature is too memory and processor intensive, it is not available on the .NET CF. To interact with COM objects a wrapper written in eMbedded Visual C++ needs to be provided, which is generally used to write DLLs for Windows CE.NET devices. Also, special care needs to be taken of not invoking features of Windows CE.NET that are unsupported on the Pocket PC.

There are times when manual *Marshaling* (serialization), may be needed in the .NET CF due to control crosses from managed to unmanaged boundaries. *Marshalling* is required whenever an object needs to be transformed to a data format that is suitable for storage or transmission. The framework offers no automatic *Marshaling* for COM components except for those provided by *P/Invoke*.

The framework only supports one API calling convention called *WINAPI*, which is used by the Windows API for specific platforms.

XML

XML is supported on the .NET CF by the *XmlReader* and *XmlWriter* classes. In order to use these classes, the *System.Xml* namespace must be imported.

XmlReader is a class that only offers a forward motion reading of XML documents as a sequence of nodes; each node representing an element, attribute, text value or other component of the document. The Boolean value returned by *XmlTextReader* indicates whether the end of the document has been reached.

If other non-forward navigation is required, DOM functionality that is offered by the *XmlDocument* class needs to be included. While the class is supported in the .NET CF, due to performance and memory considerations this is not encouraged.

XmlWriter class is used to provide forward-only and non cached XML file or stream output. The *XmlTextWriter* class can also be used, besides writing XML data; it also has support for namespaces and translating special characters into text entities. In addition, XML data can be exactly specified where it should be written to. The *Indentation* property may also be used to make XML documents easier to read.

Functions of XML that are not supported in .NET CF:

- XmlDataDocument Class
  Relational and hierarchical view of a XML document

- *XML Path Language (XPath)*
  Query over unstructured XML data

- *Extensible Stylesheet Language Transformation (XSLT)*
  Transform XML data to other format

- *XML Validation (XmlValidatingReader)*
  Validate XML correctness of a XML document

3.3.3 WCF support

A subset of WCF is included into .NET CF version 3.5 [23]. The framework does not include the *Service Model*, which handles service hosting and calling, but only message-level WCF in which messages need to be crafted manually as serialized objects. Also, it will only support the *basicHttpBinding*, with optional transport and message level security.

A command-line tool will be included that will automatically generate a proxy class for easy calling WCF services running on desktops.

3.4 Alternative .NET Frameworks

The Microsoft .NET Framework has only been created for the Windows platform. Due to this constraint, the open-source community has implemented several other alternative .NET Frameworks that also operates on Linux, Solaris, Mac OS X and Unix.

3.4.1 Mono

The Mono project [24] overcomes the single biggest shortcoming to using .NET, the requirement to run on the Windows platform. The project brings the shared source release of .NET to multiple platforms and then builds an open source project around extending it.

The Mono Framework provides the necessary software to develop and run .NET client and server applications on Linux, Solaris, Mac OS X, Windows and Unix. The framework is almost fully functional and features nearly complete implementations of *ASP.NET*, *ADO.NET* and *Web Forms*, along with almost all of the *System* namespace [25].

Following is a short list of Mono-features [24]:

- Multi-platform
- Based on the ECMA/ISO standards
- Runs ASP.NET and Winforms applications
- Can run .NET, Java, Python and more

Performance

A performance comparison between Mono and .NET has been included in appendix 10.1. The performance test comparison shows an overall of 17.77% performance lead for .NET. However, the results between the tests do vary greatly and in two cases the results show a significant performance advantage for Mono. During the *nested loop* and *exception* tests the results respectively show a 617.59% and 114.93% performance differences and thereby favoring Mono over .NET for these kinds of operations.

3.4.2 .NET CF Alternatives

Besides alternatives for the .NET Framework, there are also several open source alternatives available for the .NET Compact Framework.

OpenNETCF

OpenNETCF [26] is an open-source alternative for .NET CF which was started as an independent source for Compact Framework development information working under the spirit of the open-source movement.

The OpenNETCF Compact Framework Library offers many enhancements such as support for WSE2.0 features that provide support for the WS-Addressing, WS-Security, and WS-Attachments specifications. Also included are enhancements for debugging and tracing as well as added graphics controls.

DotGNU Portable.NET

DotGNU Portable.NET [27] includes support for compiling and running C# and C applications that use the BCLs, XML, and *Systems.Windows.Forms*.

Whereas OpenNETCF only targets the Windows CE platform, Portable.NET supports several CPUs and OS's:

- CPU: x86, PPC, ARM, PA-RISC, S/390, IA64, Alpha, MIPS and SPARC.
- OS: GNU/Linux , *BSD, Cygwin/Mingw32, Mac OS X, Solaris and AIX.

Communication

While computing begins to move away from desktop computers toward Internet enabled devices, such as hand-held computers and cell phones, many distributed applications depend on the Internet. As the vision of .NET is globally distributed systems, the Internet plays an important role. Several Internet standards are used to create a foundation for the .NET Framework to accomplish this vision.

Two of these standards will first be introduced that are essential to WSs and UPnP (HTTP and XML). Following, the concept of *Remote Procedure Calls (RPCs)* and asynchronous WS calls, and several important communication technologies will be described. Finally, this section will conclude with a communication review comparison.

3.5 HTTP/HTTPS

Hypertext Transfer Protocol (HTTP) is the standard for communication over the Internet and defines a set of rules for transferring files (text, graphic images, sound, video, and other multimedia files). According to the *Open Systems Interconnection (OSI)* seven layer model: HTTP is categorized as an application layer protocol running over TCP/IP.

HTTPS: is a *Uniform Resource Identifier (URI)* scheme syntactically identical to the HTTP: scheme. The scheme is used for normal HTTP connections, but adds an encryption layer of SSL/TLS to protect traffic. SSL is especially suitable for HTTP, as it can provide some protection even if only one side of the communication is authenticated; with most HTTP Internet transactions, generally only the server side is authenticated.

3.6 XML

Extensible Markup Language (XML) is the universal format for storing, carrying and exchanging data on the Web. XML was designed to describe data and to focus on what data is; it is a cross-platform, software and hardware independent tool for transmitting information. Due to these characteristics, the standard will be important to the future of the Web and be the most common tool for data manipulation and data transmission.

Like *HyperText Markup Language (HTML)*, XML uses tags and attributes, although they are not globally defined to their meaning but interpreted within the context of their use. Moreover, as XML tags are not predefined, custom tags need to be provided, while *Document Type Definition (DTD)* or XML Schema is used to describe data.

3.7 Object distribution

The following section is based on information provided by [28].

Within the SOA and TEAHA context, independent services are offered to consumers that are generally provided by objects located on remote systems or devices. Invoking methods or services on remote objects are called *Remote Procedure Calls (RPCs)*. The concept of RPC will be explained in more detail in the following section, followed by a few technologies that facilitate RPC, including SOAP, *Web Services (WSs)* and .NET Remoting.

3.7.1 Remote Procedure Call

A *Remote Procedure Call (RPC)* is a call of a procedure on objects, located on remote devices. RPC extends conventional local procedure calling and enable the construction of distributed, client-server based applications. The calling and called procedures may be on the same or on different network-interconnected systems.

By using RPC, the details of the interface with the network can be avoided. As RPC is transport independent, the application is isolated from the transport layer, which allows the application to use a variety of transport protocols.

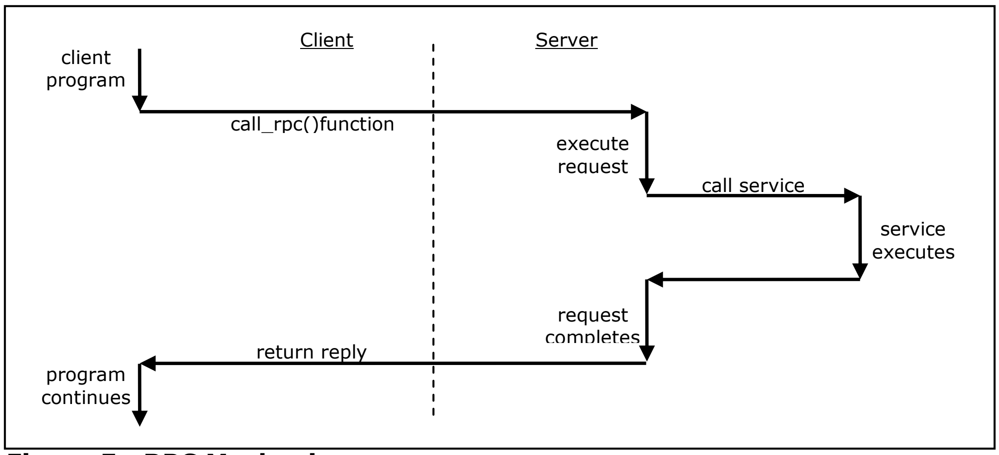


Figure 5: RPC Mechanism

The above figure shows the activity flow during a RPC. The client program initiates a procedure call that sends a service request to the remote server. The client calling thread is blocked from processing until either a reply is received from the server, or a timeout has occurred. When the server receives the request, it calls a dispatch routine that performs the requested remote service and the result is send to the client. After the RPC has completed, the calling thread is resumed and the client continues program execution.

The creation and dispatching of a PRC is handled by so called *stubs*, which are proxy objects that implement interfaces identical to the interfaces offered by the remote objects. The *client stub* intercepts method calls and is responsible for packing call parameters into a request message, after which the message is send to the server. At the server side, the message is delivered to the *server stub,* which unpacks and dispatches the message and calls the actual function on the object.

3.7.2 CORBA

The *Common Object Request Broker Architecture (CORBA)* [29] is a platform independent specification that describes how to access remote objects. With CORBA, objects can be discovered and accessed through the interfaces defined by the *Object Management Group (OMG) Interface Description Language (IDL).* The layer between clients and objects is called the *Object Request Broker (ORB)*. Within CORBA the *client stub* is called *stub*, while the *server stub* is called *skeleton*.

3.7.3 Java RMI

Java *Remote Method Invocation (RMI)* is a Java *Application Programming Interface (API)* allowing Java applications to call methods on objects, running on remote virtual machines. As Java RMI is part of the *Java 2 Standard Edition (J2SE)* most existing Java programs are able to use RMI. Java RMI relies on marshalling (serialization) of objects over the network. Similar to CORBA, the *client stub* is called *stub* while the *server stub* is called *skeleton*.

3.7.4 DCOM

Distributed Common Object Model (DCOM) is a low-level extension of the standard *Component Object Model (COM)*, used in the Microsoft Windows system. Comparable to CORBA, DCOM also uses proxies to offer RPC. The protocol used for connecting the proxies and stubs is called *Object Remote Procedure Call (ORPC).* Contrary to CORBA and Java RMI, within DCOM the *client stub* is called *stub* and the *server stub* is referred to as *Proxy*.

3.7.5 SOAP

Simple Object Access Protocol (SOAP) is a lightweight and language neutral communication protocol that allows programs to communicate via standard Internet HTTP. SOAP defines the use of *Extensible Markup Language (XML)* and HTTP to execute RPCs, and is becoming the standard for RPC over the Internet and used for accessing a WS.

Using the Internet's existing infrastructure, SOAP can work with firewalls and proxies. SOAP can also use SSL for security, and HTTP's connection management facilities [30].

```
<?xml version="1.0"?>
<SOAP:Envelope
Xmlns:SOAP="http://www.w3.org/2001/12/SOAP-envelope"
SOAP:encodingStyle="http://www.w3.org/2001/12/SOAP-encoding">
<SOAP:Header>
 ...
</SOAP:Header>
<SOAP:Body>
 ...
  <SOAP:Fault>
   ...
  </SOAP:Fault>
</SOAP:Body>
</SOAP:Envelope>
```

Figure 6: SOAP Message

3.7.6 XML-RPC

XML-RPC is a specification and a set of implementations that allow software, independent of the OS and its environment, to make procedure calls over the Internet. It uses HTTP as transport mechanism and XML for the encoding of RPC. XML-RPC is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned [31].

The procedure parameters of the RPCs can be of the following types: int, double, string, boolean, dateTime.iso8601, base64, array or struct. An overview of these types accompanied with corresponding examples is available at [32].

Java

Apache XML-RPC [33] is a Java implementation that is compatible with the XML-RPC specification, while allowing users to enable several vendor extensions. These features are only available, if a streaming version of Apache XML-RPC operates on both sides. Both server and client have a Boolean property *enabledForExtensions* in their respective configuration, for enabling these extensions:

- All primitive Java types are supported, including long, byte, short, and double.
- Calendar objects are supported.
- DOM nodes, JAXB objects or objects implemented with the *Java.io.Serializable* interface can be transmitted.
- Both server and client can operate in a streaming mode, which preserves resources much better than the default mode that is based on large internal byte arrays.

OSGi

Oscar's OSGi includes a bundle [34] that provides an XML/RPC service, which other bundles in the OSGi environment can use to register their XML/RPC handlers.

The OSGi bundle provides a *servlet bridge* from the standard OSGi HTTP Service and the Java Apache XML/RPC implementation, which has been discussed in the previous section. Furthermore, the Apache XML/RPC implementation is included as an embedded jar within the bundle.

.NET Framework

XML-RPC.NET is a library for implementing XML-RPC services and clients in the .NET environment. The library package and several examples on accessing the library are available at [35]. In addition, the library is CLS-compliant and can be called from any CLS-compliant language such as C# and VB.NET, and includes the following features:

- Interface based definition of XML-RPC servers and clients
- Code generation of type-safe client proxies
- Support for .NET Remoting
- ASP.NET WSs that support both XML-RPC and SOAP
- Client support for asynchronous calls
- Client support for various XML encodings and XML indentation styles
  (Some other XML-RPC implementations only accept certain indentation styles)
- Built-in support for XML-RPC Introspection API on server

- Dynamic generation of documentation page at URL of XML-RPC end-point
- Support for mapping XML-RPC method and struct member names to .NET-compatible names
- Support for Unicode XML-RPC strings
- Support for optional struct members when mapping .NET and XML-RPC types

.NET Compact Framework

The XML-RPC.NET distribution contains an assembly named *CookComputing.XmlRpc.CF.dll*, which provides support for the .NET CF. Because .NET CF does not support reflection, it is necessary to implement XML-RPC.NET proxies manually. However, the assembly has been marked as an experimental version and is mostly untested.

3.8 Web Services

This section is based on information provided by [36], [37] and [38].

XML *Web Services (WSs)* are small units of code designed to handle a limited set of tasks and enable applications to offer their services over a network using Internet technologies. They are main building blocks in the .NET programming model and are language and platform independent and based on open protocols such as HTTP, SOAP, and XML.

Large websites providing WSs, such as eBay and Amazon, extend APIs that essentially turn its Web site into a platform. The WSs they provide are available to developers to build applications that can connect to those services. The growth and use of APIs across the Web illustrate the rapid growth of WSs.

WSs still miss many important features like security and routing that are defined in the WS specifications. Although, these important features will be supported once SOAP becomes more advanced.

3.8.1 Web Services Architecture

WSs support HTTP access remotely or locally on the device, and are a simplified way of exposing services across network computer or server to connect to a Web browser that supports XML. When a call is made to a WS, the server returns a response in XML.

WSs can be accessed using standard Web formats (HTTP, SOAP, and XML), without needing to know how the WS is implemented. Official Web standards (XML, UDDI, and SOAP) are used to describe what Internet data is, what WSs can do, and help users to locate specific services.

An important characteristic of WSs is that they have a document/procedure-oriented architecture instead of an object-oriented architecture. They do not support an object reference-model; all data in a document is passed by value.

WS Components

There are three major components that make up a WS:

- The WS on the Server side
- The client application calling the WS via a Web Reference
- A WSDL WS description, describing the functionality of the WS

A variety of specifications are associated with WSs that may complement, overlap, and compete with each other. These specifications may relate to topics such as directory access, service description, messaging and function calls and security specifications. The WS specifications are also occasionally referred to as WS-*, for the reason that many specifications use "WS-" as prefix for their names. An overview of several categorized specifications is available at [39].

Benefits and future

By using WSs, existing services are encapsulated and allow applications to publish their functions or messages on the Internet. Furthermore, they offer important SOA characteristics such as interconnectivity and reusability. Also, unlike other remote access methods, WSs are able to cooperate with firewalls without requiring special setup or diminishing the effectiveness of firewalls. They simplify the communication between different applications and provide an easy way to offer services and distribute information to a large number of consumers.

An application calling a WS will send its requests using XML, and get its answer returned as XML. The calling application will therefore never be concerned about the underlying programming language or OS running on the other computer; data can be exchanged between different applications and different platforms while developers are able to reuse existing services instead of writing new ones. Using XML based communication protocols the independency of both OS and programming language is assured and increases the interoperability, interconnectivity and reusability of WSs.

Following the SOA principles of reusability and composability, a new development of Web-based applications called *mash-ups,* mix at least two different services from different, competing Web sites to compose new services. A mash-up could for example overlay traffic data from one source on the Internet over maps from Microsoft or Google. This capability to mix and match data and applications from multiple sources into one dynamic entity is considered to represent the promise of the WS standard.

3.8.2 WSDL

Web Services Description Language (WSDL) documents provide the description of a WS interface. WSDL is based on XML and is used to define WSs, describe how to access them, and to specify the location of the service and the methods it exposes.

The WSDL defines the restrictions on the format of the SOAP messages, which are used by a WS client to communicate with a WS provider. A top-down approach is typically used for developing WS clients, to generate code from a WSDL. Whereas a bottom-up approach, is used for developing WS providers and to generate a WSDL from code.

A WSDL document describes a WS using these major elements:

Element	*Defines*
<types>	*The data types used by the WS*
<message>	*The messages used by the WS*
<portType>	*The operations performed by the WS*
<binding>	*The communication protocols used by the WS*

Table 1: WSDL Elements

- **Types**
  Defines the data type that is used by the WS. For platform neutrality, WSDL uses XML Schema syntax to define data types.

- **Message**
  Defines the data elements of an operation, and can consist of multiple parts, which are similar to *function call parameters* in traditional programming language.

- **PortType**
  This is the most important WSDL element and defines a WS, allowed operations, and messages that are used. The port defines the exposed WS interfaces and the connection point to a WS. It can be compared to a function library in traditional programming languages, while each operation can be compared to a function.

- **Binding**
  Defines the message format and protocol details for each port.

There are several operation types available, for defining in the *PortType* section. WSDL defines the following operation types, including the most common *request-response* type:

Type	*Definition*
One-way	*The operation can receive a message but will not return a response*
Request-response	*The operation can receive a request and will return a response*
Solicit-response	*The operation can send a request and will wait for a response*
Notification	*The operation can send a message but will not wait for a response*

Table 2: WSDL Operation Types

Given the previous major elements, the main structure of a WSDL document is as follows:

```
<definitions>
  <types>
    definition of types...
  </types>

  <message>
    definition of a message...
  </message>

  <portType>
    definition of a port...
  </portType>

  <binding>
    definition of a binding...
  </binding>
</definitions>
```

Figure 7: WSDL Structure

A WSDL document can also contain other elements, like extension- and a service- element that offers grouping of definitions of several WSs in one single WSDL document.

[40] provides a full example of a WSDL Document and several examples of *one-way* and *request-response* operations. Furthermore, it describes how to define the communication protocol used by the WS and bind to SOAP.

3.8.3 Interaction

A client interacts with a WS through SOAP packages, which are generated and processed by a client proxy. The WSDL document that describes the target WS is needed to create the proxy.

First, the client proxy receives a service request from the client, after which the proxy processes the request and serializes it into a corresponding SOAP request package. Following, the SOAP package is forwarded to the remote WS, which dispatches and executes the requested method. A SOAP response package, containing the results from the method call, is then send back to the client proxy, which finally deserializes the SOAP package and forwards the actual method results to the client.

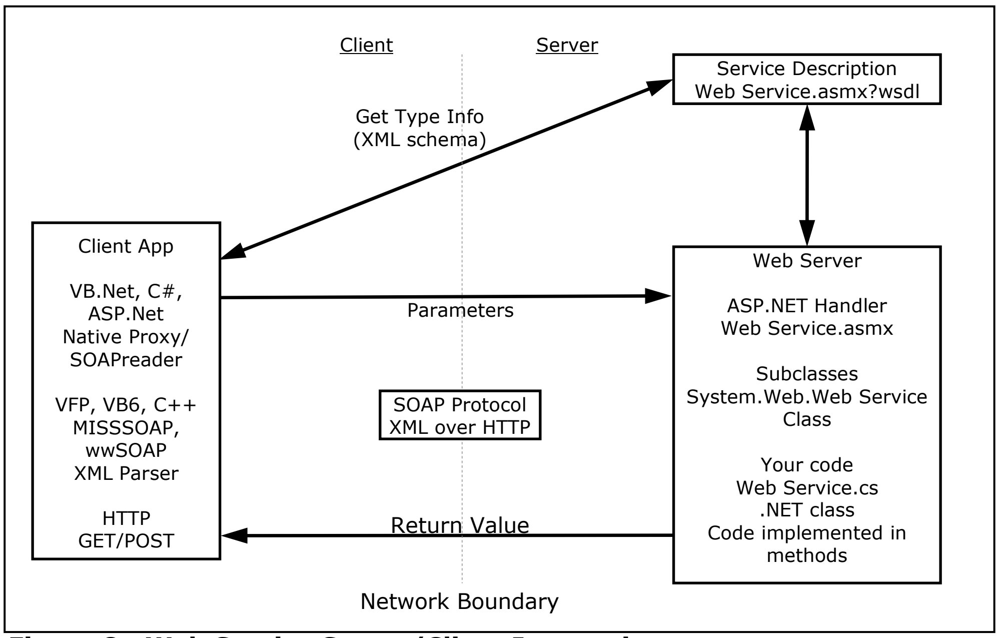


Figure 8: Web Service Server/Client Interaction

3.8.4 Asynchronous Calls

This section is based on information provided by [41].

An asynchronous call to a WS, allows the calling thread to continue executing while it waits for the WS to respond. The call is made on a different thread than the one that is running the user interface, which allows users to continue interacting with an application without it locking up.

When calling WSs asynchronously, multithreaded programming techniques are required to avoid problems when multiple threads try to access the same data simultaneously. Within the C# .NET environment, a common method for obtaining exclusive access to an object is the *lock* statement. Another solution is to use the *Monitor* class and *Control.Invoke* to make applications thread-safe.

Calling WSs asynchronously is a two-step operation. For every synchronous method in the proxy class, there is a corresponding *Begin* and *End* method. For example, if the name of the WS method is *GetItems*, the asynchronous methods would be *BeginGetItems* and *EndGetItems*. The *Begin* method initiates the Web method call, while the *End* method completes the call and retrieves the WS response.

Conversational

A WS is *conversational* if the series of messages transmitted between the two endpoints are tracked with a unique conversation ID, with specific operations flagged to start and end the flow of messages. WSs that provide callbacks are, by definition, conversational.
The initial operation initiated by the client and the callback operation made by the WS are related to each other and must be tracked with a unique conversation ID. If this is not done, there is no way for the client to differentiate between callback operations relating to different initiating calls.

Callbacks

Clients may need WSs to respond to a client defined endpoint. These callback operations require the WS clients to provide a callback endpoint that is capable of asynchronously receiving and processing callback operation messages.

The following diagram shows the main entities involved, in case a client uses a WS with a callback operation:

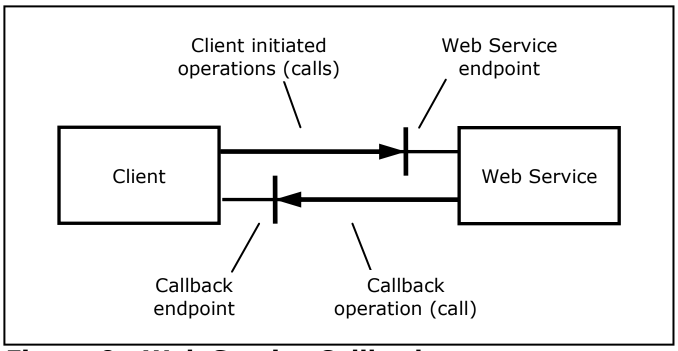

Client initiated operations (calls)
Web Service endpoint
Client
Web Service
Callback endpoint
Callback operation (call)

Figure 9: Web Service Callback

Because callbacks are separated from the original request, to which the callback is a response, they appear as unsolicited messages. Many hosts refuse unsolicited network traffic, because they reject the unsolicited message or are protected by firewalls. Clients that run in such environments are therefore not capable of receiving callbacks.

Another requirement for handling callbacks is that the client is persistent by being conversational. If the client is a web application, or a non-conversational WS, it cannot handle callbacks.

Polling

To avoid the complications of having to provide callback endpoints, a technique known as polling may be used as alternative.

In order to allow clients that can't accept callbacks to use WSs, a polling interface can be supplied as an alternative. In a polling interface, one or more methods are provided that a client calls periodically to determine if the result of a previous request is ready.

Although the WS or Java control will still be asynchronous in design, the interactions with a client are handled with synchronous, unbuffered methods. A guideline for implementing WS polling interfaces has been given in [42].

However, this technique requires the client to periodically call the server to check for callback events. The overhead of these calls can therefore be significant if the event does not occur frequently [43].

3.8.5 WS-Eventing

This section is based on information provided by [44].

WS-Eventing defines a protocol for WSs to subscribe, or to accept a subscription from another WS. WSs often want to receive notifications when certain events occur in other services and applications.

A mechanism for registering events is required because the set of WSs interested in receiving those event messages is often unknown in advance, and will most likely change over time. The WS-Eventing specification defines a protocol for a WS (*subscriber*) to subscribe to another WS (*event source*) in receiving *event messages*.

Currently WS-Eventing is only supported on the .NET CF using external code [45].

3.8.6 WS-Policy

This section is based on information provided by [46].

The WS-policy specification allows WSs to use XML for advertising their policies, and for WS clients to specify policy requirements. WS-policy is a set of specifications that describe the capabilities and constraints of the security and business policies on intermediaries and end points, and how to associate these policies with services and end points.

WS-Policy provides a single policy grammar, and defines a policy as a collection of one or more policy assertions. Some of the assertions specify traditional requirements and capabilities, such as the authentication scheme and transport protocol selection, while others specify requirements such as privacy policy and *Quality of Service (QoS)*.

WS-PolicyAttachment

Defines two general-purpose mechanisms for associating policies with the subjects to which they apply. This specification also defines how these general-purpose mechanisms can be used to associate WS-Policy with WSDL and UDDI descriptions.

WS-PolicyAssertions

This specification defines a common set of policy assertions for WSs. The assertions defined by the specification cover text encoding, natural language support, versioning of specifications, and predicates.

3.8.7 Framework Support

Many frameworks include support for hosting and consuming WSs, and allow an easy implementation of WS functionality in client and server applications. However, if WS support is not available, offering WSs from a web page can be easily achieved by including support for reading and generating XML documents.

Instead of clients generating SOAP requests, there is a much simpler alternative for calling WSs. With REST techniques [47] a request can simply be plain URL parameters on a HTTP GET, instead of bulky complex SOAP messages. In either case, the result of the request operation will always be the return of an XML document.

.NET

The .NET Framework has support for creating and consuming WSs. Furthermore, the proxy class that Visual Studio .NET generates when a Web Reference is added, includes methods for synchronously or asynchronously accessing the WS.

However, WS support on .NET CF is limited and does not allow hosting but only the calling of WSs. In addition, large parts of the WS-* suite, including security related and the WS-Discovery specification are also not supported. Moreover, several examples on accessing WSs from the .NET CF are available at [41].

Java

There are several technologies that offer Java support for enhanced, secure, legacy and core WSs [48]. They enable development of enhanced WSs using features such as reliable messaging and atomic transactions, and provide solutions for the processing of XML content, data binding and the development of SOAP based and REST-ful WSs.

OSGi

The Knopflerfish Axis port provides SOAP/WS access to any OSGi bundle, both for exporting OSGi services as WSs and for importing WSs into an OSGI framework [49].

The A*xis-OSGi bundle* offers the ability to export services, registered on the OSGi framework service registry, as SOAP services. The exported objects must only expose data types supported by SOAP. In order to export a service object as a SOAP service, it only requires the *SOAP.service.name* property to be set on the registered service.

The Axis-OSGi bundle depends on an installed and started *commons-logging* bundle. Furthermore, the JRE must support XML or be provided with an XML parser. Client and WSDL related Axis classes are optional; depending on the need for Axis clients on the OSGi framework, or necessity of server-generated WSDL for the SOAP services.

3.8.8 WSE

The *Web Services Enhancements (WSE)* for Microsoft .NET [50] and [51] is an add-on for .NET Framework and Visual Studio that enables developers to develop secure, interoperable WSs based on open industry specifications.

The add-on implements the WS-* specifications to provide benefits such as end-to-end message level security, content-based routing, and policy by leveraging WS-Addressing, WS-Security, and WS-Policy.

WSE introduces a set of *turnkey security assertions*, securing common SOAP message exchange patterns. Instead of applying security to SOAP messages on a per SOAP message basis, these security assertions are designed to be applied to SOAP message exchanges that are based upon the distributed application's scenario.

Furthermore, WSE enables developers to use declarative files to specify behavior, including security requirements, when receiving and sending messages from a client or a service. The files consist of a collection of named policies, each of which defines a set of requirements, including security for a SOAP message exchange.

Additionally, WSE 3.0 is wire-level-compatible with WCF, using the HTTP protocol and the corresponding *turnkey security scenarios* [52]. However, interoperability is not guaranteed with other protocols such as TCP.

3.9 .NET Remoting

This section is based on information provided by [53].

.NET Remoting uses the .NET concept of an *Application Domain (AppDomain)*, which is an abstract construction for ensuring isolation of data and code. A process may contain multiple AppDomains, while an AppDomain can only exist in exactly one process.

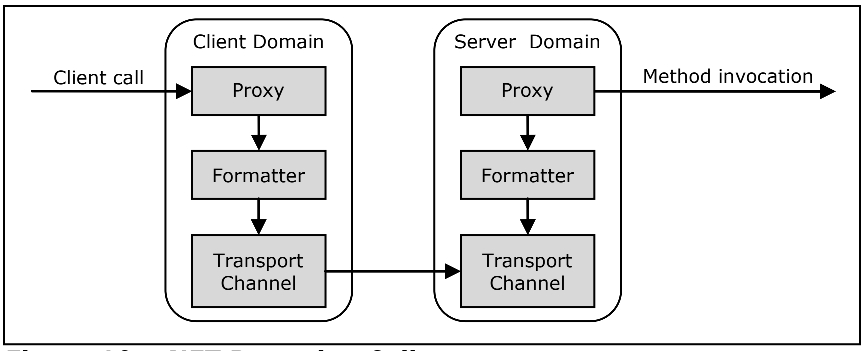


Figure 10: .NET Remoting Call

A client uses a proxy object to invoke public remote methods on a remote object. A remote object class always inherits from *System.MarshalByRefObject*, as it provides the core foundation for enabling remote access of objects across AppDomains.

Two formatters are offered by .NET Remoting: binary and SOAP. The former is fast and encodes method calls in a proprietary, binary format. The latter is slower, due to increased package sizes and complexity, but encodes method calls in an open format. Additionally, a custom formatter may be provided if neither do suffice.

The lower-layer *Channels* are used to physically transport the messages to and from remote objects. The standard included *TcpChannel* and *HttpChannel* can be extended, or if the existing channels do not suffice, a custom channel can be used.

3.9.1 Remote object

The remote object can be defined as *client-activated* or *server-activated*.

A client-activated object is created and destroyed by the client, and will be subject to garbage collection once it is determined that no other clients need it. Server-activated objects are managed by the remote server, and are only created when a client actually invokes a method on the proxy.

Additionally, server-activated objects can be defined as:

- *Singleton,* stateful object that retain state across multiple method calls.
- *Single-call object,* handles only a single client request, where each call is made on a new object instance and no state is held between calls.

3.9.2 DCOM

Previously, DCOM was used to handle interprocess communication between applications. As DCOM relies on a proprietary binary protocol that is not supported by all object models, enabling interoperability across platform is complicated. In addition, DCOM uses a range of ports that are generally blocked by firewalls, and prohibits easy usage over the Internet.

.NET Remoting removes the difficulties of DCOM by supporting a variety of transport protocol formats and communication protocols, which allows it to be adaptable in many different network environments.

3.9.3 .NET CF support

.NET Remoting is not natively supported by the .NET CF, however [54] introduces a design for bridging .NET CF and .NET Remoting.

Additionally, [55] provides several .NET Remoting coding examples. The coding examples show how to create a Remotable Object, the server that exposes the object and a Remoting client that access the Remotable Object on the server.

3.9.4 Java support

Java offers no native functionality for accessing .NET using .NET Remoting. However, the specification that governs .NET Remoting was released to *Ecma International* as part of the CLI specification [56], resulting in several available commercial products that offer Java Remoting bridging. These Remoting engines generate proxies that allow .NET applications to invoke Java/J2EE systems, and Java/J2EE to invoke .NET applications [57].

3.10 WCF

This section is based on information provided by [58].

Windows Communication Foundation (WCF), formally named Indigo, is a communication subsystem to enable applications to communicate using the network. It is one of the four major API's in .NET Framework 3.0 that unifies Microsoft communication technologies: WSs, .NET Remoting, Distributed Transactions and Message Queues into a single SOA programming model for distributed computing. It is intended to provide *Rapid Application Development (RAD)* to WSs, with a single API for inter-process communication in a local machine, *Local Area Network (LAN)*, or over the Internet. WCF runs in a *sandbox* and provides the enhanced security model that all .NET Framework applications provide.

When a WCF process communicates with a non–WCF process, XML-based encoding is used for the SOAP messages. However, when the process communicates with another WCF process, the SOAP messages are encoded in an optimized binary format. Both encodings conform to the data structure of the SOAP format, called *XML Information Set (Infoset)*.

In .NET CF, the WCF programming model support will not include service model, but only channel layer messaging support. This means the program will send and receive messages and it will be up to the application to correlate messages with each other, as there are no RPC semantics.

3.10.1 Channels

The actual sending and receiving of messages to a network resource is performed by *Transport Channels*. Whereas *Layered Channels* perform a function based on the input message and delegate further modification and transmission to other channels in the channel stack; examples include Protocol Channels that use message headers and infrastructure messages to establish a higher-level protocol, such as WS-Security.

Furthermore, all necessary extensibility points are exposed to allow custom transport and layered channels to be used, while maintaining a unified programming model.

3.10.2 Performance

There is a clear performance advantage when using WCF compared with .NET Remoting and ASP.NET WSs [59]. Although the performance tests show a distinct performance advantage for WCF, they were conducted on an IIS hosted WSs and .NET Framework system environment.

The test results might therefore not represent the performance-level on a client device using the .NET CF. Moreover, the tests do not take disk space or memory usage into account, which are important software characteristics on resource-constrained devices, such as routers and portable devices.

3.11 Sockets

A socket is a unique end-point of a communication link between two programs running on an IP-based network, allowing applications to read and write from/to the network.

To send a message over a socket, the target machine's IP address and the process identifier of the application is needed. The process identifier of an application is a unique number, which is also referred to as *port*.

3.11.1 Raw socket

Raw sockets are sockets that allow access to packet headers on incoming and outgoing packets and are usually used at the transport or network layers.

Usually raw sockets receive packets with the header included as opposed to non-raw sockets, which strip the header and receive just the payload. They are not a programming language-level construct, but are part of the underlying OS networking API. Most socket interfaces are based on the BSD socket interface and therefore support raw sockets.

3.11.2 .NET and Java Sockets

In .NET, *System.Net.Sockets.Socket* can be used as a socket in server and client applications, and allows both synchronous and asynchronous operations. Java provides the Java.NET package, which offers the *Socket* and *ServerSocket* class that respectively implement the client and the server side of the connection.

Furthermore, several coding examples are available at [60] that illustrate how server and client programs can read from and write to sockets.

3.12 Communication Comparisons

This section includes several comparisons of the communication technologies introduced in the previous sections, and provides an overview of the benefits and drawbacks. However, the prime focus of attention will be on .NET Remoting and WSs, as they are the main .NET communication technologies.

3.12.1 SOAP and XML-RPC

This section is based on information provided by [61].

SOAP and XML-RPC are specifications that are both intended for remote procedure calling. While SOAP is a *World Wide Web Consortium (W3C)* standard, XML-RPC is proprietary and frozen; however both specifications are platform independent.

XML-RPC excels in being lightweight, but is often too simple for enterprise usage and there is less agreement on error messaging. SOAP is more complex, but also more extensible and has better support for complex data expressiveness. Additionally important communication characteristics such as security, authentication, and encryption are also supported with help of WS-Security or XML security standards.

Furthermore, a new specification that starts to emerge is REST, which handles requests through URL parameters on a HTTP GET instead of the bulky messages used by SOAP and XML-RPC.

3.12.2 Web Services and OSGi Services

The key difference between WSs and OSGi services is that WSs always require some transport layer, which makes it remarkable slower than OSGi services that use direct method invocations. Also, OSGi components can directly react on the appearance and disappearance of services [7].

3.12.3 Web Services and .NET Remoting

This section is based on information provided by [53].

Although both .NET Remoting and WSs can enable cross-process communication, they are designed to support different audiences. WSs are part of .NET Remoting, but have a simplified programming model and are intended for a wide target audience, while .NET Remoting provides a more complex programming model and has a much narrower reach.

The following table comprises a comparison overview of both technologies.

	ASP.NET Web Services	**.NET Remoting**
Protocol	Can only be accessed over HTTP.	Can be accessed over any protocol (including TCP, HTTP and SMTP)
State Management	Work in a stateless environment.	Supports stateful and stateless environments through Singleton and SingleCall objects.
Type System	Supports only data types defined in XSD type system, limiting the number of serializable objects.	Supports rich type system, by using binary communication.
Interoperability	Support interoperability across platforms, and are ideal for heterogeneous environments.	Requires the client be built using .NET, enforcing homogenous environment.
Reliability	Highly reliable due to the fact that WSs are always hosted in IIS.	Takes advantage of IIS for fault isolation. If IIS is not used, application needs to provide plumbing to ensure application reliability.
Extensibility	Offers extensibility by allowing to intercept SOAP messages during serialization and deserialization.	Very extensible by customizing the different components of the .NET Remoting framework.
Ease-of-Programming	Easy-to-create and deploy.	Complex to program.

Table 3: Web Services VS .NET Remoting

Serializing

WSs serialize objects using XML and can only handle items that can be fully expressed in XML. .NET Remoting relies on the existence of the CLR assemblies that contain information about data types. This limits the information that must be passed about an object, and allows objects to be passed by value or by reference.

State Management

WSs are stateless and they handle each incoming request independently. Also, each time a client invokes an ASP.NET WS, a new object is created to service the request and is eventually destroyed after the method call completes. To maintain state between requests, session and Application objects can be used. However, maintaining state can be costly with WSs as they use extensive memory resources.

.NET Remoting supports a range of state management options and has three types of remote objects, as opposed to one with WSs. The ability to mix and match the various object types facilitates creation of solid architectural designs, and more efficient, scalable applications.

Performance

Comparing to the performance of .NET Remoting with a SOAP formatter, the performance of ASP.NET WSs is better. The usage of XML can cause SOAP serialization to be many times slower than a binary formatter. Additionally, string manipulation is very slow when compared to processing of the individual bits in a binary stream. Therefore, .NET Remoting provides a clear performance advantage over ASP.NET WSs when TCP channels with binary communication are used.

Security

.NET Remoting does not provide support for securing cross-process invocations. However, a Remotable Object hosted in IIS can access all the same security features provided by IIS. If the TCP channel is used, or the HTTP channel is hosted in a container other than IIS, authentication, authorization and privacy mechanisms need to be provided.

ASP.NET WSs hosted in IIS, benefit from IIS features such as support for secure SSL communication, authentication and authorization services.

Type Fidelity

ASP.NET WSs favor the XML schema type system and offer a simple programming model with broad cross-platform reach. .NET Remoting favors the runtime type system and offers a more complex programming model with a much more limited reach.

Reliability

.NET Remoting allows hosting of remote objects in any type of application including Windows Forms, managed Windows Services, console applications and the ASP.NET worker process. The ASP.NET worker process holds all AppDomains in which every instance of an ASP.NET application is created.

If remote objects are hosted in a Windows service, or a console application, features like fault tolerance within the hosting application are necessary to safeguard the reliability of the remote object. However, when remote objects are hosted in IIS, advantage can be taken of the fact that the ASP.NET worker process is both auto-starting and thread-safe. As for ASP.NET WSs, they are always hosted in IIS, and take advantage of IIS security abilities, therefore reliability is not an issue.

Extensibility

Both the ASP.NET WS and .NET Remoting infrastructures are extensible. Inbound and outbound messages can be filtered, and aspects of type marshaling and metadata generation can be controlled. Additionally, .NET Remoting extensibility allows the implementation of custom formatters and channels.

Furthermore, as ASP.NET WSs rely on the *System.Xml.Serialization.XmlSerializer* class for marshalling data to and from SOAP messages, the marshaling can be easily customized by adding a set of custom attributes for controlling the serialization process. This offers fine-grained control over the XML being generated when an object is serialized.

Service Discovery

Computing power is ever more being added to smaller, more common portable devices, while they are extended with connectivity and networking that is easy to use and configure. *Service Discovery (SD)* protocols play an important role in providing the latter.

There are many SD protocols such as WS-Discovery, Jini, Salutation, *Simple Lookup Protocol (SLP)* and Bluetooth *Secure Discovery Protocol (SDP);* one of the SD protocols that has gained much popularity is *Universal Plug and Play (UPnP)*.

UPnP offers easy access and discovery of distributed multimedia files, devices and services on the network and thereby offering the consumer a better product experience. The protocol is based on Internet protocols and is simple enough for being implemented on small appliances, while also being powerful enough to be scaled to the global Internet.

One of TEAHA's main objectives is the development of advanced residential gateway subsystems. For this purpose, the OSGi framework with UPnP support has been proven to be a suitable foundation. Although there are many SD protocol available, the focus of attention in this chapter will be mainly on UPnP, seeing that the TEAHA framework also relies on UPnP for discovering TEAHA devices and the services they provide.

3.13 Jini

Jini technology [62], currently developed under project Apache River [63], follows the principles of SOA and defines a programming model that both exploits and extends Java to create secure, distributed systems consisting of network services and clients. It can be used to build adaptive network systems that are scalable, evolvable and flexible, as generally required in dynamic computing environments.

The SOA is Java-based and adds distribution to the JVM, and offers a number of capabilities such as SD and mobile code. Jini combines Java with *look-up services* and a method for discovering services to provide SD. Furthermore, it relies on the transportation of Java interfaces over the network; if a device wants to use a remote service, it will receive a proxy that forwards the interactions to the service.

The Jini surrogate technology, which is built upon Jini and the Java platform, allows small devices to upload a small piece of Java code to another device. Using this code, the device can intermediate and act as a translator to allow services from small devices to be offered on the Jini network.

3.14 Web Service Discovery

There are several standards available for discovering WSs. This section will introduce the UDDI directory service and the WS-Discovery multicast SD protocol, the latter is intended for ad-hoc networks with a minimum of networking services.

3.14.1 UDDI

Universal Description, Discovery and Integration (UDDI) is a directory service where businesses can register and search for WSs. It is a platform-independent framework for describing services, discovering and integrating business services by using the Internet. UDDI communicates via SOAP and uses Internet standards such as XML, HTTP, DNS protocols and it also uses WSDL to describe interfaces to WSs

3.14.2 WS-Discovery

This section is based on information provided by [58] and [64].

Web Services Dynamic Discovery (WS-Discovery) defines a multicast SD protocol that enables advertisement and dynamic discovery of services on ad-hoc and managed networks. The specification is part of the WS-* suite and relies on other WS specifications to provide secure, reliable, transacted message delivery and to state WS and client policy.

The protocol is not intended to support Internet-scale discovery, to provide extended information on services, to define a data model for service description or rich queries over that description. It is meant to support discovery of services in ad hoc networks with a minimum of networking services (e.g., no DNS or UDDI directory services), and enable smooth transitions between ad hoc and managed networks.

In order to scale to a large number of endpoints, the protocol defines the *multicast suppression behavior* if a discovery proxy is available on the network. To minimize the need for polling, services that wish to be discovered send an announcement when they join and leave the network. Clients can query for services by type as well as scope without heavy administrative costs. Search messages are sent to a multicast group, if services match the search query they return responses directly to the requester.

3.15 UPnP

This section offers the reader insight on the basic workings and usage of the *Universal Plug and Play (UPnP)* standard. It is based on and inspired by several official documents offered by the UPnP community available at [65], [66], [67], [68] and [69].

UPnP is a distributed, open networking architecture, intended for peer-to-peer network connectivity of wireless devices, intelligent appliances, home entertainment equipment and computers. It is designed to bring easy-to-use, flexible, standards-based connectivity to ad-hoc or unmanaged networks and to simplify the implementation of networks in the home and corporate environments.

In addition to control and data transfer among networked devices, UPnP enables seamless networking by defining and publishing UPnP device control protocols, built upon open, Internet-based communication standards. These standards are able to span different physical media, and enable multiple-vendor interoperation, and synergy between the Internet and home or office intranets. Further, via bridging, UPnP accommodates media and connected devices that are running non-IP protocols, when reasons of cost, technology or legacy prevent them from running IP.

The architecture is designed to support zero-configuration, "invisible" networking and automatic discovery for a wide range of device categories. This means that a device can use the UPnP defined common protocols to dynamically join a network, obtain an IP address, convey its capabilities on request and learn about the presence and capabilities of other devices. It enables data communication between any two devices under the command of any control device on the network. Furthermore, it allows a device to leave a network smoothly and automatically without leaving any unwanted state behind.

Because UPnP uses no device drivers and is defined by the common protocols it uses, it is independent of OS, programming language and physical medium. It does not specify the APIs applications will use, allowing OS vendors to create APIs that will meet their customer needs. Thereby, allowing vendor control over device *User Interface (UI)* and interaction using the browser as well as conventional application programmatic control.

- **Media and device independence**
  UPnP technology can run on any medium including phone lines, power lines (PLC), Ethernet, IR (IrDA), *Radio Frequency (RF)* - (Wi-Fi, Bluetooth), and FireWire. No device drivers are used; common protocols are used instead.

- **Common base protocols**
  Base protocol sets are used, on a per-device basis.

- **User interface control**
  UPnP allows vendor control over device UI and interaction using a web browser.

- **OS and programming language independence**
  Any OS and programming language can be used to build UPnP products.

- **Internet-based technologies**
  UPnP technology is built upon IP, TCP, UDP, HTTP, and XML, among others.

- **Programmatic control**
  UPnP architecture also enables conventional application programmatic control.

- **Extendable**
  Each UPnP product can have value-added services layered on top of the basic device architecture by the individual manufacturers.

3.15.1 Devices, Services and Control Points

The UPnP architecture can be divided into three elements: devices, services and control points. The figure below gives a context overview of these elements that will be discussed in the following section.

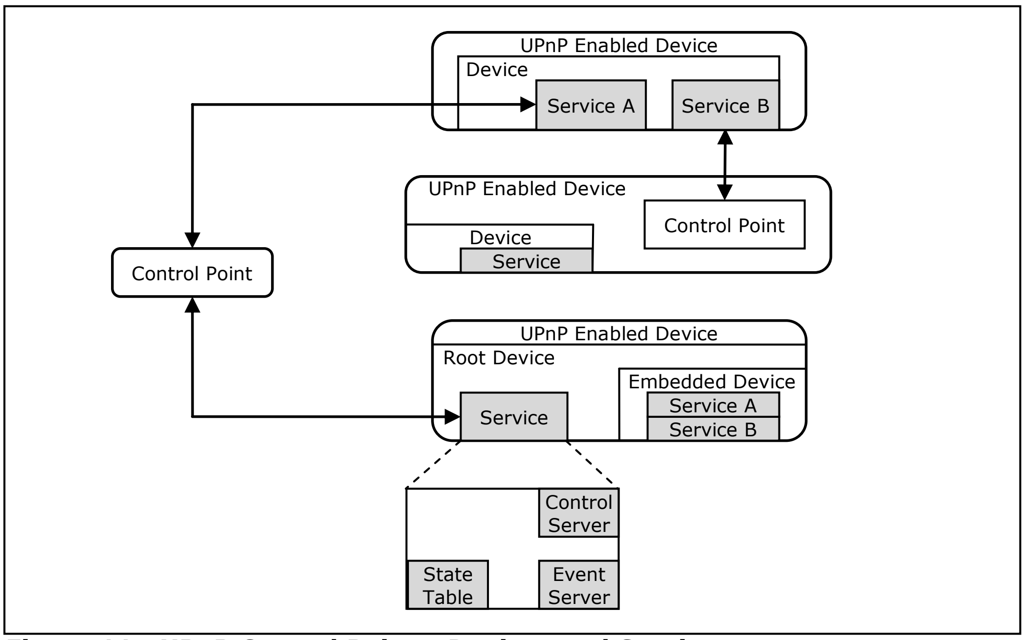


Figure 11: UPnP Control Points, Devices and Services

Devices

A UPnP device is a container of services and may also include nested devices. For instance, a TV/VCR combo offers TV services and also contains a nested VCR device.

Different categories of UPnP devices will be associated with different sets of services and embedded devices. Consequently, different working groups standardize on the set of services a particular device type provides. This information is captured in an XML device description document that is hosted by the device. In addition to the set of services, the device description also lists properties associated with a device, such as the device name and icons.

Services

The smallest unit of control in a UPnP network is a service. A service exposes actions and models its state using state variables. For instance, a clock service could be modeled as having a state variable *current_time*, which defines the state of the clock, and two actions *set_time* and *get_time*, which allow control of the service.

Similar to the device description, service information is part of an XML service description that is standardized by the UPnP forum. The device description document contains an URL pointer to these service descriptions.

A UPnP service consists of a state table, control server, and an event server. The state table models the state of the service through state variables and updates them when the state changes. The control server receives action requests, such as *set_time*, executes them, updates the state table and returns responses. The event server publishes events to interested subscribers anytime the state of the service changes. For instance, a fire alarm service would send an event to interested subscribers when its state changes to *ringing*.

Control Points

A control point is a controller capable of discovering and controlling other devices in a UPnP network. After a control point has discovered another device, it is able to:

- Retrieve the device description and get a list of associated services.
- Retrieve service descriptions for interesting services.
- Invoke actions to control a service.

- Subscribe to a service’s event source, to receive an event from the event server whenever the state of the service changes.

3.15.2 UPnP Protocol Stack

The next figure shows the UPnP protocol stack, which consists of higher layers defined by UPnP vendors, working committees and the device architecture. The lower layers provide necessary services to the higher layers and are based on several open Internet standards.

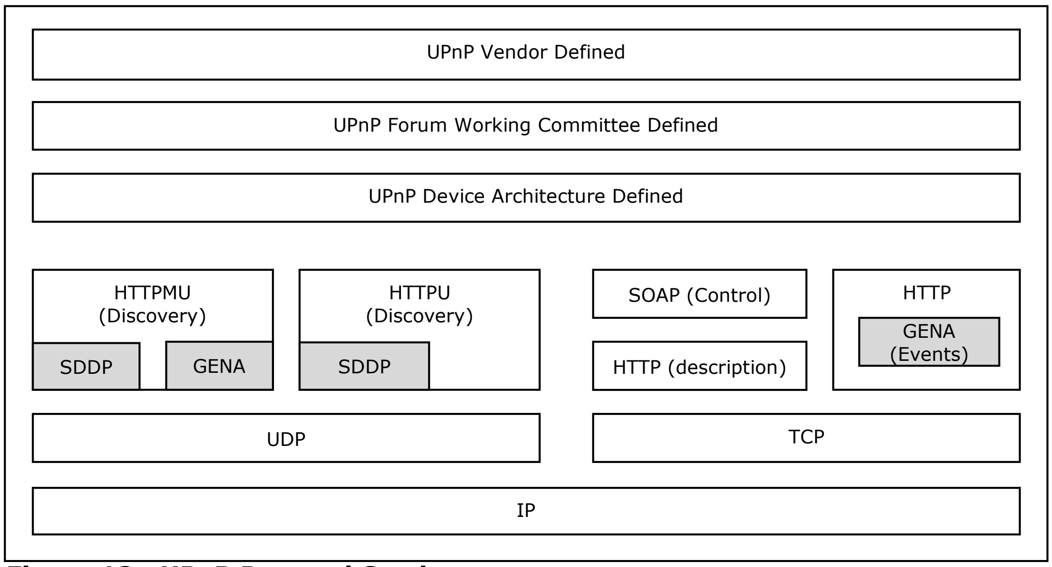


Figure 12: UPnP Protocol Stack

The protocol stack accommodates UPnP devices to perform the following actions:

- Respond to search requests issued by control points.
- Parse SOAP and HTTP control commands from control points and return the responses to control points.
- Submit events to subscribing control points.
- Publish information about the device and its services and nested devices.
- Respond to device queries from control points.
- Respond to control requests from control points.

Next, the protocol stack will be discussed in a top-to-bottom approach.

3.15.2.1 ***Vendor, Committee and Architecture Defined Layers***

UPnP vendors, UPnP Forum Working Committees and the UPnP Device Architecture document define the highest layer protocols used to implement UPnP. Vendors can deliver UPnP on a variety of hardware platforms and physical networks that support IP.

The UPnP Device Architecture layer defines a schema or template for creating device and service descriptions for any device or service type.

The UPnP Forum templates, created by UPnP Forum working committees on top of the UPnP Device Architecture, define domain- and device-specific meanings and the format of data. Various device and service types are standardized and a template is created for each individual device or service type.

The UPnP vendor-specific layer contains the application, user interface, and vendor-specific hardware. Vendors add their own extensions on top of the aforementioned Forum templates and fill them in with device or service specific information, such as device name, model number, manufacturer name and the URL service description. This data is then encapsulated in UPnP-specific protocols, defined in the UPnP Device Architecture document, such as the XML device description template.

3.15.2.2 *Lower Protocol Layers*

The network media, the TCP/IP protocol suite and HTTP provide basic network connectivity and addressing. On top of these open, standard, Internet based protocols, UPnP defines a set of HTTP servers to handle discovery, description, control, events, and presentation. Moreover, UPnP uses HTTP over *User Datagram Protocol (UDP)*, known as HTTPU and HTTPMU, for unicast and multicast communication.

Networking Media for UPnP

Devices on a UPnP network can be connected using any communication media, including wireless RF, phone and power line, IrDA, Ethernet and IEEE 1394. However, other technologies could also be used to network devices together like HAVi, CeBus, LonWorks, EIB or X10. These technologies can participate in the UPnP network through a so called UPnP bridge or proxy.

TCP/IP

The *Transmission Control Protocol (TCP) / Internet Protocol (IP)* protocol stack serves as the base for the UPnP protocol, and enables network connectivity between UPnP devices. By using the standard TCP/IP protocol suite, UPnP is able to span different physical media, and ensure interoperability with multiple vendors.

UPnP devices can use many of the protocols available in the TCP/IP stack, including TCP, UDP, IGMP, ARP and IP; as well as services such as DHCP and DNS. Since TCP/IP is one of the most ubiquitous networking protocols, it is relatively easy to create or locate an implementation for a UPnP device that is adjusted for footprint and performance.

HTTP, HTTPU and HTTPMU

HTTP is a core part of UPnP as all aspects of UPnP are built on top of HTTP. HTTPU and HTTPMU are variants of HTTP, defined to deliver messages on top of UDP/IP instead of TCP/IP. The basic message formats used by these protocols are similar with that of HTTP and is required both for unicast and multicast communication, and when message delivery does not require reliability and the overhead associated with it. These protocols are used by SSDP, which will be discussed next.

SSDP

Simple Service Discovery Protocol (SSDP) defines how network services can be discovered on the network. SSDP is built on HTTPU and HTTPMU and defines methods for a control point to locate network resources, and for devices to announce their availability on the network.

Similarly, a device connecting to the network will send out multiple SSDP presence announcements, advertising the services it supports. Both presence announcements and unicast device response messages contain an URL location pointer of the device description document, which has information on the set of device properties and services.

By defining the use of both search requests and presence announcements, SSDP eliminates the overhead that would be necessary if only one of these mechanisms is used. As a result, every control point on the network has complete information on network state while keeping network traffic to a minimum.

In addition to the discovery capabilities, SSDP also provides a way for a device and associated services to gracefully leave the network and includes cache timeouts for purging old information for self healing.

GENA

Generic Event Notification Architecture (GENA) was defined to provide the ability to send and receive notifications using HTTP over TCP/IP and multicast UDP. GENA formats are used in UPnP to create the presence announcements to be sent using SSDP and to provide the ability to signal changes in service state for UPnP eventing.

GENA defines the concepts of subscribers and publishers of notifications to enable events. A control point, interested in receiving event notifications, subscribes to an event source by sending a request that includes the service of interest, a location to send the events to and a subscription time for the event notification. The subscription must be renewed periodically and can be canceled using GENA.

3.15.2.3 *UPnP Function Layers*

UPnP devices provide services that can be initiated by the device itself or by a control point. Some objects on the network can be both a control point and a device, which is the case for most home audiovisual devices.

The previous section discussed the UPnP protocol in terms of its protocol layers. In this section, UPnP will be discussed in terms of the six basic function layers that offer the services to create SD functionality.

The protocol is reviewed in a bottom-to-top approach, starting with the lower three layers that exist in all devices and control points, followed by three optional higher layers.

The six basic UPnP function layers:

1. IP addressing
2. Device discovery
3. Device description

4. Action invocation or control
5. Event messaging
6. Presentation or user interface

The lower function layers (1, 2, and 3), the higher layers (4, 5, and 6) are optional.

Control points (layer 4) can initiate an action on a device. Most devices will have event messaging (layer 5) but not control or presentation; they will create an event message, while a control point listens for these event messages.

Some devices might just provide a user interface to the control point (layer 6). The control point will display the user interface of the device, for displaying events, status or controlling the device. Since every UPnP device has a Web server, the browser in a control point can be used as the front panel of the device. The presentation layer is required because a pointer to the presentation URL is part of the device description. However, this layer is not required if the control point is handling the device programmatically and not through a Web browser.

Addressing

The addressing layer handles the assignment of IP addresses to control points and devices. The first thing a device needs to do is to obtain an address in order to connect to the network. The IP addresses can come from a *Dynamic Host Configuration Protocol (DHCP)* server, from a range of addresses handled by Auto IP, or be hard wired.

A DHCP server assigns IP addresses to devices from a user defined range. Each device must have a DHCP client and search for a DHCP server when it is initially connected to the network. If the DHCP client does not get a response from a server after a time-out, it will retry to make sure the server has a chance to respond.

If the network does not contain a running DHCP server, the client uses Automatic-IP addressing to choose an appropriate address. Auto-IP assigns an IP address from a set of reserved private addresses. The address must then be tested to determine if the address is already in use. Otherwise another address must be chosen and tested, up to an implementation dependent number of retries. If the address is assigned using Auto-IP, the device will periodically check if a DHCP server becomes available on the network, to ensure connectivity is maintained among devices.

At this point, the device either has a DHCP, or an Auto-IP assigned address. In either case, the device can communicate with other devices on the network using TCP/IP. Once the device has a valid IP address for the network, it can be located and referenced on that network through that address. If during the DHCP transaction, the device obtains a domain name through DNS, the device should use that name in subsequent network operations, or otherwise use its own IP address.

Discovery

The discovery layer is where control points search for UPnP devices on the network, or UPnP devices advertise their presence. In both cases the discovery message contains a few, essential specifics about the device or its services, such as its type, identifier and an URL pointer to its XML device description document.

Search

When a control point is added to the network, it multicasts a SSDP discovery message using HTTPMU to search for devices and services available on the network. The control point can refine the search to find only devices of a particular type, services or device. These search messages contain vendor-specific information, such as device or service types and identifiers.

All devices must listen to the standard multicast address for these messages. If any of their embedded devices or services matches the search criteria in the discovery message, they will send a unicast SSDP response to the control point using HTTPU. These responses contain the same information as discovery advertisements and are sent to the IP address of the control point that initiated the search.

Advertisement

Whenever a UPnP device connects to the network, it will send GENA advertisements for each embedded device and service, announcing its presence through multicast HTTPMU. Any interested control point can listen to the standard multicast address for notifications when new services are available. Since these messages are sent using UDP, an unreliable transport, they could be resend several times to ensure they are received correctly.

The discovery messages include a time stamp to indicate how long the advertisement is valid. Before this time expires, the device must resend its advertisement. The device should also send a message to tell the network it is going away before disconnecting.

Description

Once a control point discovers a device, it still knows little about the device. To inform about the device and its capabilities, or to interact with the device, the control point must retrieve the device's description from the URL contained in the discovery message.

To retrieve a UPnP device description, the control point issues an HTTP GET request on the description URL. Also, service descriptions can be retrieved in the same way, using service description URLs that are part of the device description.

The UPnP device description is an XML document that contains several pieces of vendor-specific information, definitions of all embedded devices, presentation URL of the device, and an enumeration of all services, including their URLs for control and events.
It also contains information regarding device type, manufacturer, icons, serial number, model name and number, product code, and other similar information.

Device types are defined by the UPnP Forum and can have one or more UPnP templates to define the content and presentation of data. Additionally, devices can contain other, logical devices and services.

For each service, the service description includes a list of commands or actions, and the parameters or arguments for each action to which the service responds. In addition, the service description also includes a list of variables that models the state of the service at run time. The state variables are described in terms of their data type, range, and event characteristics.

Control

After a control point retrieves a description of a device, actions can be sent to a service provide by the device. Control messages are sent to the control URL of a service, and any effects of the action are modeled by changes in the state variables. Also, control points can query state variables of a service, but only a single variable with each query sent.

The UPnP Forum working committee defines the action names, argument names and variables contained in control messages. This information is encapsulated in UPnP specific formats and formatted using SOAP.

A device must respond to control requests within 30 seconds. The response could indicate that the action is still pending, while an event will signify completion and return any results or errors.

Event Messaging

Within the event messaging layer, control points can listen for notifications of UPnP device state changes. Because there can be multiple control points and UPnP enabled devices on a network, *eventing* was designed to keep all control points equally informed about the effects of any action. As a result, state variables that are described in a service description can all be *evented*.

To get event messages, control points subscribe to event messages for a particular service by sending a subscription message. The receiving device may accept the request and respond with a duration for the subscription. Once subscribed, the subscribed devices are able renew the subscription during that period, or cancel the subscription if it is no longer interested.

If the event occurs, the service sends an event message to all current subscribers of the event. Furthermore, when a device leaves the network gracefully, it will send an advertisement that it is leaving. The sub- and un-subscribing of events allow control points to listen to events in a selective way.

Event messages use GENA, which is defined to send and receive notifications using HTTP over TCP. Event messages contain the names of one of more state variables and the current value of those variables.

A special initial event message is sent when a control point first subscribes, and contains the names and values for all evented variables and allows the subscriber to initialize its model of the state of the service.

Presentation

The presentation layer is used to give users control over the UPnP device, and requires the completion of the first three layers: getting an address, discovering the device, and obtaining the device description.

The presentation page URL is contained in the device description. After issuing an HTTP GET request to the URL, the device will return a presentation page that can be loaded into a web browser, and allow a user to control the device and/or view device status. There are no constraints on the use of the page, while an unlimited number of linked presentation pages are permitted.

The capabilities of the presentation page are completely specified by the UPnP vendor; there is no UPnP Forum element defined in presentation. To implement a presentation page, the vendor may wish to use UPnP mechanisms for control and/or events, leveraging the device's existing capabilities.

However, HTML-based management has its limitations, as it has no simple proper way to asynchronously report status changes to clients. Therefore, these changes are reported using event messaging and control points.

3.15.3 Security

This section is based on information provided by [70], [71] and [72].

Regular UPnP

Regular UPnP offers no facilities for encryption or authentication and falls short in providing a secure way for communication. Using UPnP as such, might therefore allow eavesdropping and unauthorized control of network devices.

UPnP does not have a lightweight authentication protocol, while the available security protocols are complex. As there is no authentication or authorization used in UPnP, every device on the network is able to control another UPnP device.

Therefore, using UPnP does introduce some security issues. Especially UPnP enabled routers are a security threat to the network as they can be exploited by malicious software. Local computers that are infected with malicious software might send UPnP commands to the router and open specific ports to the Internet. Opening ports could possibly allow outsiders to gain network control or access to local computers. Nevertheless, when UPnP services are used in a controlled and trusted environment, the security risks of using UPnP can be minimized.

Secure UPnP Version 1

UPnP defines a suite of protocols that enable home networking without a traditional network administrator. UPnP V1 Security defines a pair of secure services: *DeviceSecurity* and *SecurityConsole*.

- *DeviceSecurity* needs to be added to any device that needs to be access-controlled.
- *SecurityConsole* needs to be offered by the administrative application or device at which the homeowner makes security decisions known to the home network.

The UPnP V1 standard categorizes components into the following classes:

- Devices - Offering services and holding resources
- Control Points - Discover Devices (SSDP), request services (SOAP), receive event notifications (GENA).

Additionally, UPnP V1 Security, it adds a third component, the *SecurityConsole*, which is both a Device and a Control Point. It discovers all security-aware components, lets the user to name them, and allows the user to:

- take or share ownership, edit a device's *Access Control List (ACL)*, of user's devices
- edit ACL of any owned device, grant authorizations to certain Control Points
- (optionally) generate certificates for defining named groups of control points or for delegating authorization

Each security-aware component has its own RSA key pair and a Device or Control Point is known to the Security Console by its public key hash. For the ease-of-use, the user is able to name any component in his/her personal network and use that name instead.

The basic architecture of UPnP V.1 is client-server, with the client called a *Control Point* and the server called a *Device*. As mentioned before, UPnP uses three protocols to let components interact with each other: SOAP, SSDP, and GENA.

SOAP is secured by allowing only authorized Control Points to invoke any secured action within a Device. This is accomplished by an ACL in each secured device, each of the entries lists a unique Control Point ID, a name of a group of Control Points, or the universal group *<any/>*, and the actions that are allowed on the device.

SSDP is a protocol that is regarded as difficult to secure. It is vital for authorized users to discover other network devices, but it is also desirable that an attacker is not able to take an inventory of the network devices. Therefore, to secure SSDP, any existence of Devices are announced as *generic Devices*, and are not described in any detail except in response to requests from authorized Control Points. However, even if all traffic was completely encrypted, an attacker would still be able to take an inventory of a home network just based on timing and length of messages.

3.15.4 Intel UPnP stack

Intel has provided a UPnP stack that is available with a royalty free license to copy, modify and create derivative works of the source code, and which operates on the .NET and .NET CF. An overview of the Intel UPnP stack and a device implementation is included in appendix 10.2, while the Intel UPnP stack and an evaluation toolkit is available at [73].

Security

Secure communication is very important within distributed environments. The .NET bundle will therefore also need to provide support for policy enforcement, authentication and encryption; depending on the level of security that is required. In this section some important security concepts, XML security standards and the *Diffie-Hellman (DH) secret key exchange protocol* will be described.

3.16 Security Concepts

Following, several important basic key concepts will be described that are essential to providing secure communication [74].

Authentication
The process of verifying an identity claimed by or for a system entity, which may be services, clients or central directories.

There are several ways that authentication can be used: *client* or *server authentication*. The former requires clients to authenticate to use a service, while the latter requires Service Agents and Directory Agents to prove their identity to clients. Another possibility is that endpoints have to authenticate to each other. Achieving authentication include username and password methods and stronger ones such as digital signatures.

Authorization
The right or permission granted to an entity, to access a system resource. Achieving authorization include the usage of *capabilities* or *ACLs*. Credentials can also be used for establishing either a claimed identity or the authorizations of a system entity.

Confidentiality
The property that information is not disclosed to an unauthorized entity. Usually encryption, such as public and symmetric key encryption, is used to achieve confidentiality.

Trust
Means the extent to which someone, who relies on a system, can have confidence that the system meets its specifications. Protocols may use certificate authority-based trust models that provide public key infrastructures to secure transactions.

Integrity
The property that data has not been changed, destroyed, or lost in an unauthorized or accidental manner. Message integrity can be achieved through verification mechanisms, such as hashes.

Non-repudiation
A security service providing protection against false denial of involvement in a communication. For non-repudiation purpose, data hashes may be used and signed before encryption.

Privacy
Privacy is the right of an entity to determine the degree to which it will interact with its environment, including the degree to which it is willing to share information about itself.

3.17 Digital Signatures

Digital signatures, as used in SSL, consist of a private and public part that enables authenticated communication. The public part of a device certificate (X.509) is available to all network devices and includes a public key that is used to authenticate and verify the data integrity of messages.

The signature of a message, a *keyed-Hash Message Authentication Code (KHMAC)*, is generated by encrypting the hash of the message with the private key. Accordingly, the message integrity can be verified by decrypting the signature using the public key and comparing the result with the computed message hash.

Moreover, certificates must be signed by a *Trusted Third Party (TTP)*, such as a *Certificate Authority (CA)* to allow verification of the certificate itself. The arrangement that binds public keys with respective user identities by means of a CA is referred to as a *Public Key Infrastructure (PKI)*.

However, caution is required when generating and choosing device certificates. Several popular hash methods are subject to collision attacks and should not be used in certificate generation, as they enable public keys and valid certificates to be crafted for misabuse, as described in [75].

While, the hash methods can still be safely used for generating HMACs [76], it is still advisable to omit using hash methods that are subject to collision attacks in order to protect against any security vulnerabilities that may arise in the future.

3.18 XML Security standards

WS and UPnP communication are both based on the SOAP XML standard. Some of the important XML-based security specifications include the following [38]:

XACML
eXtensible Access Control Markup Language (XACML) is a framework for defining a set of privileges required to perform an operation, including identity information and external factors, like access policy and time of day.

XML digital signature
XML Signatures (XML DSIG) provide integrity, message authentication and/or signer authentication services for data, whether located within the XML that includes the signature or elsewhere.

XML Encryption
Offer data confidentiality using encryption techniques. Encrypted data is wrapped inside XML tags defined by the XML Encryption specification. A deliverable has been made available that defines how to encrypt (portions of) an XML document.

XKMS
XML Key Management Specification (XKMS) consists of two parts:

- *XML Key Information Service Specification (X-KISS)*
  Defines a protocol for a trust service that resolves public key information contained in *XML-SIGelements*. The protocol allows a client of such a service to delegate part or all of the tasks required to process elements.

- *XML Key Registration Service Specification (X-KRSS)*
  Defines a protocol for a WS that accepts registration of public key information.
  The public key may be used in combination with other WSs, including the abovementioned X-KISS.

SAML
Secure Assertion Markup Language (SAML) is an XML-based framework for exchanging identification information, user authentication, entitlement, and attribute information.

The framework allows entities to make assertions on the identity, attributes, and entitlements of a subject to other entities. A *Trusted Third Party (TTP)* could provide a signed set of assertions identifying an identity.

SAML allows partner applications to share user authentication and authorization information, which is essentially the *Single Sign-On (SSO)* feature offered by all major vendors' e-commerce products.

In the absence of any standard protocol on sharing authentication information, vendors generally use HTTP cookies to implement SSO. With the advent of SAML, this data can be wrapped inside XML, so that cookies are not needed and interoperable SSO is possible.

3.19 Web Services

Security has become a hot topic for WSs, it is important for WS security to address topics such as access control, authentication, data integrity and privacy. WS technology has been moving towards different XML-based security schemes.

In the WS context, *security* means that a message recipient will be able to do some or all of the following:

- Verify the *integrity* of a message.
- Receive a message *confidentially*.
- Determine the *identity* of and authenticating the sender.
- Determine if sender is *authorized* to perform the operation in request message.

In a distributed environment these requirements are met by using cryptography. Two of the most fundamental security operations, signing and encrypting, can directly meet the first two needs. The other two requirements, and all supporting infrastructure that they need, are built on top of those operations.

3.19.1 Security Measures

The following list [77] depicts common classes of attacks and prevention the mechanisms:

- Message alteration
  Prevented by including signatures of the message information using WS-Security.

- Message disclosure
  Confidentiality is preserved by encrypting sensitive data using WS-Security.

- Address spoofing
  Prevented by ensuring that all address are signed by an authorized party for the address.

- Key integrity
  Maintained by using the strongest algorithms possible, by comparing secured policies.

- Authentication
  Established using the mechanisms in WS-Security and WS-Trust. Each message is authenticated using the mechanisms in WS-Security.

- Accountability
  Function of the key type, strength of the key and algorithms that are being used. Strong symmetric keys in general provide sufficient accountability. However, in some environments, strong *Public Key Infrastructure (PKI)* signatures are required.

- Availability
  Reliable messaging services are subject to availability attacks. Replay detection is a common type of attack that is to be addressed by the mechanisms described in WS-Security and/or by caching message identifiers. Moreover, care should be taken to ensure that minimal state is saved prior to any authenticating sequences.

- Replay
  Mechanisms should be used to identify replayed messages such as the timestamp/nonce outlined in WS-Security. Alternatively, other technologies, such as sequencing, can also be used to prevent replay of application messages.

3.19.2 WS-Security

WS-Security specifies how to sign and encrypt SOAP messages. At the simplest level, it says to put the XML DSIG signature in a SOAP Header element with a specific name:

```
<SOAP:Envelope xmlns:SOAP='http://schemas.xmlSOAP.org/SOAP/envelope/'>
  <SOAP:Header>
    <wsse:Security xmlns:wsse='http://schemas.xmlSOAP.org/ws/2002/07/secext'>
      <Signature xmlns='http://www.w3.org/2000/09/xmldsig#'>
        ...defined below...
      </Signature>
    </wsse:Security>
  </SOAP:Header>
  <SOAP:Body id='Body'>
     ...message body...
  </SOAP:Body>
</SOAP:Envelope>
```
Figure 13: WS-Security

Unfortunately WS-Security only allows one instance of this header, making multiparty signatures (for example an online auction with buyer, seller, and auctioneer) a little awkward. Nevertheless, if the tamper-evident proof for the application message is kept, only the SOAP message need to be kept.

The Signature contains three parts: a number of references to what is being signed, a signature value that covers the references, and information about the signing keys.

The *SignedInfo* element mainly contains references to data, a description of how to process and generate a hash for, and what that hash (message digest) value is.

```
<SignedInfo>
  <CanonicalizationMethod Algorithm='http://www.w3.org/TR/2001/REC-xml-c14n-20010315'/>
  <SignatureMethod Algorithm='http://www.w3.org/2000/09/xmldsig#rsa-sha1'/>
  <Reference URI='#Body'>
    <Transforms>
      <Transform Algorithm='http://www.w3.org/TR/2001/REC-xml-c14n-20010315'/>
    </Transforms><DigestMethod Algorithm='http://www.w3.org/2000/09/xmldsig#sha1'/>
    <DigestValue>riJUygbyupbDqcIiV+jgIdHe7WQ=</DigestValue>
  </Reference>
</SignedInfo>
```
Figure 14: SignedInfo

This fragment states that the signed data is canonicalized and then hashed and signed with an RSA key. The node with a "Body" ID attribute is XML data that is canonicalized and has the specified hash value.

The signature value itself is a hash of the *SignedInfo* element, it is important to realize, however, that XML DSIG gains its flexibility by always doing an indirection; the data itself isn't signed, a reference that specifies the data's digest is.

```
<SignatureValue>fHP...ZOA=</SignatureValue>
```
Figure 15: Signature

Finally, we have information about the key that generated the signature, which in this case is an X.509 digital certificate.

```
<KeyInfo>
  <X509Data>
    <X509Certificate>MII...AABvi</X509Certificate>
  </X509Data>
</KeyInfo>
```
Figure 16: X.509 Digital Certificate

3.19.3 Web Services and SSL

This section is based on information provided by [74], [77] and [78].

Secure Sockets Layer (SSL) may be used in certain, restricted cases for securing WSs as it has a number of limitations when used with WSs.

Message and Transport-level security

The type of technology used to protect a message determines to which extent the message remains protected while it is in transition.

SSL is a point-to-point protocol operating between communication endpoints and only provides transport-level security as described in the below figure. At each step a message will be decrypted and a new SSL connection set up. Therefore, if a message goes through multiple intermediates, as is the case with WSs, it will be exposed to each intermediate.

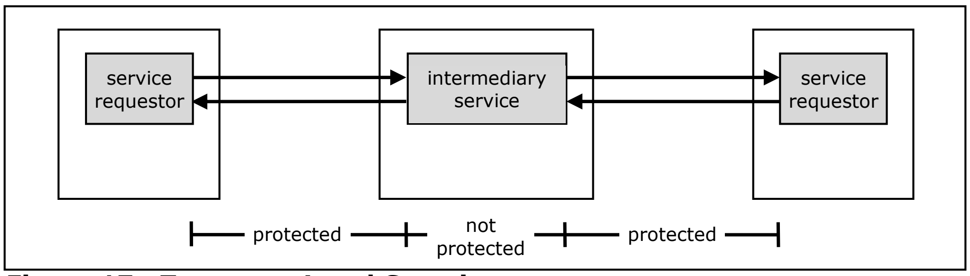


Figure 17: Transport-Level Security

To ensure that a message is fully protected along its entire message path, message-level security is required. Security measures are applied to the message itself; regardless of where the message may travel, the applied security measures go with it.

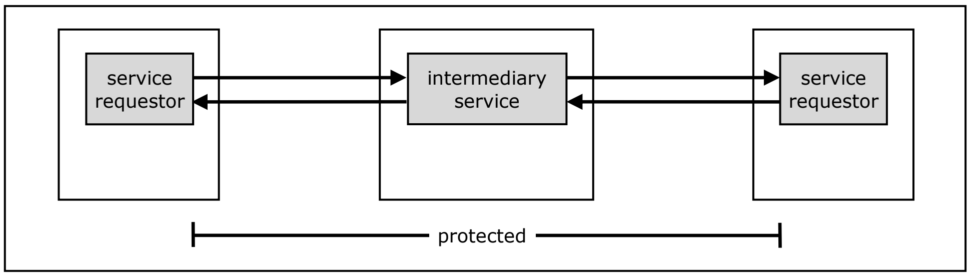


Figure 18: Message-Level Security

As a result, securing WS communication requires message-level security that operates at the same layer as the message, and which is part of the SOAP message itself. For this purpose, SOAP headers fit nicely with *detached signatures* to make secure WSs possible.

Moreover, when using SSL/TLS, the message cannot be saved for later to prove that it has not been altered; the fundamental architecture of SSL/TLS makes it impossible. If for example, a service intermediary intercepts a message, it can easily alter its contents, which is a severe vulnerability.

In addition, when a connection is established, parties exchange a transient *session key* for encrypting the communication data between them. Both parties have the same key that is only intended to be used for a short period of time. In case of a dispute, it is impossible for either party to prove that it has the unmodified message.

Also, SSL encrypts the whole message, whereas WS-security supports efficient encryption of only selected parts of a message. In addition, SSL does not support authorization, while there are WS specifications available that specifically support authorization, such as XACML.

Concluding

WS-security is more efficient, XML based, and a more matching fit with WSs. However, there are times when SSL is an alternative, and is recommended to be used in combination with WS-Security in certain cases.

3.20 Diffie-Hellman protocol

Diffie-Hellman (DH) is a widely used secret key exchange protocol that provides mutual key and entity authentication. Although, DH is vulnerable to man-in-the-middle attacks, using authentication can solve this security vulnerability.

Assumptions

- Let A and B be users that want to exchange a secret key.
- Let GF(p) be a finite field (p is a prime and can be public).

Procedure

1. A selects a secret random number Ra
2. B selects a secret random number Rb
3. A and B choose a common public number (generator), $g \in GF(p)$

4. A sends $X_A = [g^{Ra} \bmod p]$ to B
5. B sends $X_B = [g^{Rb} \bmod p]$ to A

6. A calculates $K = [X_B^{Ra} \bmod p]$
7. B calculates $K = [X_A^{Rb} \bmod p]$
8. A and B now share secret key K

3.21 Station-to-Station protocol

The *Station-to-Station (STS)* protocol is a cryptographic key agreement scheme based on DH. It entails two-way explicit key confirmation, making it an *Agreement with Key Confirmation (AKC)* protocol. In addition to protecting the established key from an attacker, the STS protocol uses no timestamps.

STS introduces authentication to DH by the addition of a function $S_X(M)$ that signs message M for user X, and requires users to be able to validate signs.

Assumptions

- Let A and B be users that want to exchange a secret key
- Let GF(p) be a finite field (p is a prime and can be public).

- Let $S_X(M)$ be a function that signs message M for user X
- Let $E_K(M)$ be the encryption function using key K
- Let C_X be the digital certificate for user X

Procedure

1. A selects a random secret number Ra
2. B selects a random secret number Rb
3. A and B choose a random public number (generator), $g \in GF(p)$

4. A sends $X_A = [g^{Ra} \bmod p]$ to B
5. B sends $[X_B, C_B, E_K(S_B(X_B, X_A))]$ to A, $X_B = (g^{Rb} \bmod p)$
6. A sends $[C_A, E_K(S_A(X_A, X_B))]$ to B

7. A calculates $K = [X_B^{Ra} \bmod p]$
8. B calculates $K = [X_A^{Rb} \bmod p]$
9. If both signs are validated A and B now share secret key K

Design and Implementation

4 Design Concepts

In chapter 3 a technology overview was given, discussing several frameworks, communication technologies and security concepts. As there are currently no existing solutions available that enable secure interaction between .NET and TEAHA, a few design concepts will be introduced in section 4.2.

Moreover, several additional resources, concepts and additional requirements will be discussed that are essential for composing a suitable design.

4.1 Design Approach

This section describes several resources and concept elements that are either of help, or which are essential for designing a suitable solution.

While taking the requirements in chapter 2 into consideration, several design concepts will be introduced and reviewed. After which, eventually, a final design can be composed that allows OSGI and .NET devices to discover and access each other's services in a secure and transparent manner.

In order to provide a final design that is suitable for a broad variety of .NET devices, the design must enclose classes, methods and external code that are available on both .NET CF and .NET Framework. Seeing that, the .NET CF is used on resource restricted mobile devices, it is primarily consisting of a subset of .NET namespaces, classes and methods. Therefore, to ensure a higher degree of compatibility, the restricted .NET CF is taken to represent the .NET platform.

OSGi Bundles

Bundles form an important part of the OSGi framework specification and allow a diversity of services to be dynamically added or removed. As there are existing bundles covering several networking standards, a selection is made of available bundles that may be of help for implementing communication between .NET and TEAHA/OSGi.

- The Domoware UPnP base driver
  Implements the UPnP base driver specification standard defined by OSGi.

- Axis bundle
  Knopflerfishs Axis servlet bundlefication, makes objects automatically available as WSs when they are registered with the *OSGi lookup* and when the property *SOAP.service.name* is set.

- XML RPC
  Provides an XML/RPC Service that bundles within the OSGi environment can use to register their XML/RPC handlers. The bundle provides a *servlet bridge* from the standard OSGi HTTP Service and the Apache XML/RPC implementation.

TEAHA and .NET Communication

For realizing communication between .NET and TEAHA, the possible solutions are restricted to using communication methods that are supported in .NET. However, as a few of these methods in sections 3.8-3.11 do not offer SD, additional protocols are required in order for devices to be able to search and discover other network devices. Moreover, SD is also of essence when a .NET device enters the network, and tries to locate and initiate communication with the TEAHA gateway.

As TEAHA relies on UPnP for discovering, controlling, describing devices and services, UPnP could be an appropriate choice. However, if WSs are used for implementing communication, WS-Discovery would instead be a more congruent choice for applying SD.

Remote proxy

The design of the .NET bundle requires an implementation of the *remote proxy pattern*. Proxies are classes that function as interfaces to another object, such as a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. A remote proxy provides a reference to an object located in a different address space on the same or different machine.

As devices and services enter and leave the TEAHA network, proxy objects are respectively created and destroyed. The proxy objects act as translating interfaces between .NET and TEAHA devices and allow interaction with .NET devices in a TEAHA manner, while TEAHA devices can be approached as regular .NET devices. [79]

Data conversion
Serialization is the process of encoding an object or class into a persistent or transportable state. Unfortunately, according to [80], streaming serialized objects between the Java and the .NET serializer is not possible due to incompatibility. Therefore, explicit data conversion of Java and .NET data will be required.

With SOAP or XML, communication will be independent of the OS and programming language. XML serialization converts data into a transferable XML stream, which can be deserialized into the original data at the receiving end. Due to the available XML serializers for both .NET and Java, a custom data conversion implementation can be avoided.

There are, however, a few disadvantages of using regular (non-binary) XML serialization, such as increased data sizes and processing overhead; data needs to be converted to size inefficient SOAP/XML. Moreover, communication on mobile devices should be bandwidth efficient, as they often depend on wireless connectivity that are considered expensive; both financially and in terms of processing and power consumption.

In addition, the exchange of certain data types introduces several serialization problems. Aside from the exchange of *primitive data*, which are components based on the underlying type system for .NET or Java, there are two other categories of data types that impose other serialization problems that are difficult to solve [80]:

- Non-existent data types
  A certain data type is only available in one architecture; there is no obvious, apparent mapping between the data types.

- Complex data types
  All the data types that differ from the common primitive ones, and are made up of numerous or nested primitive data types. The XML serialization and mapping of complex data types can be problematic and less evident.

4.2 Concepts

Having set out a suitable design approach in the previous section, several design concepts can be composed that will be introduced in the following section. The designs are based on communication technologies and protocols that have been discussed previously in the technology overview.

The introduced design concepts will all rely on the TEAHA gateway for policy enforcement on service interaction, which is described in section 5.3. Moreover, many of these design concepts rely on UPnP for including support for SD.
Although, WCF has been discussed in section 3.10 it is not included as a viable communication technology, due to the limited support on the .NET CF.

4.2.1 UPnP

TEAHA uses UPnP for discovering, controlling and describing devices and services. Therefore, UPnP could be an appropriate choice for including SD and communication support in the .NET bundle.

In comparison to raw sockets and even .NET Remoting, UPnP can be regarded as a high-level communication protocol. Several changes to the UPnP protocol are however necessary in order to comply with the security requirements stated in chapter 2.

Furthermore, an UPnP interface is defined that declares security and other communication management services. The interface is needed for offering the required security features, and to facilitate secure communication between .NET and TEAHA devices; it allows .NET clients to discover and access TEAHA services in a secure manner, while TEAHA devices are able to access .NET devices using the UPnP protocol.

OSGi support
Oscar's OSGi offers a bundle that implements a UPnP base driver, *the Domoware UPnP base driver bundle* [81], which maps UPnP devices to the OSGi Service Registry and OSGi services to the UPnP network. The bundle can be used as an appropriate base for implementing secure UPnP communication in OSGi.

.NET support
The design also requires UPnP functionality to be included in the .NET client application. For this purpose, the *Intel UPnP stack* [73] can be used that offers a UPnP implementation for the .NET Framework, as well as conveniently for the .NET CF. With help of the Intel UPnP stack, .NET devices are able to be serviced as UPnP control points, servers or renderers, and interact with other UPnP devices and services on the network.

4.2.2 Web Services

To enable communication between .NET and TEAHA using WSs, the .NET bundle has to host and expose WSs. The WS design must enable .NET devices access TEAHA devices and services using regular WS requests, while TEAHA devices are allowed to interact with .NET devices in a TEAHA manner. Moreover, a Registration WS is to be provided by the gateway that enables .NET devices and services to register to the TEAHA network in a secure way.

OSGi support
In section 4.1 an OSGi bundle is described that offers WS support, which can be used for designing and implementing the .NET bundle. The bundle enables OSGi devices to invocate WSs, and OSGi services to be exported as WSs.

.NET support
As aforementioned, the .NET CF does not support the hosting, but only consuming of WSs. However, this does not dismiss WSs as being a suitable design solution. It may suffice to only host WSs on the OSGi gateway, while .NET devices take on the role of WS clients.

Due to this fixed client-server model and missing support for WS hosting and WS-Eventing, it is important that asynchronous WS calls are possible. This type of call allows server-side initiated service requests to be sent properly to .NET clients.

Asynchronous WS calls can be implemented in .NET CF, following the procedure described in [41].

Security
Since WSs use HTTP, SSL could be used to provide secure communication on less secure public networks. However, as SSL only provides transport-level security, any intermediates in the message chain can potentially intercept the message and alter its contents. Whereas, XML-security standards on the other hand, provide message-level security that is more suitable when using WSs.

Although WSE is not natively supported on the .NET CF, WSs can be extended with WS-Security using external code [45].

Eventing
This section shows several approaches for extending WSs with eventing, which is required to enable notification of subscribed network devices when a certain service event occurs, or whenever devices and services enter or leave the network.

WS-Eventing
The WS-Eventing specification has been defined to extend WSs with eventing. Even though WS-Eventing is not natively supported on the .NET CF, an implementation of the WS-Eventing specification is given by [45].

Polling
When polling is used, the .NET client will periodically call a WS on the gateway that collects all eventing information from connected TEAHA devices. Whenever a service event occurs, the event will be cached by the WS for a predefined period of time. A subsequent polling call will return all cached instances of a particular service event.

However, as events do not occur frequently and SOAP is complex and size consuming, polling is regarded as inefficient that results in a great amount of processing overhead.

Eventing Socket

Eventing can also be implemented with dedicated eventing sockets that transport eventing data between event sources and subscribers. The eventing sockets are set up during communication, and are available during the entire session for exchanging eventing data.

However, the requirement for a free and open port can cause problems, as ports may be blocked by firewalls, used by applications, or disabled for security reasons. Furthermore, without taking additional safety measures, socket communication is unauthenticated, unencrypted and unsecure.

Asynchronous Web Services

Asynchronous WSs [81] may be used for implementing eventing, while the implementation of asynchronous WSs in .NET CF has been described in [82].

A first device issues an asynchronous WS call to an *Event Subscription Service* on the eventing device to receive certain event notifications. The service call includes subscription information, such as the subscribing device, and the event it subscribes to.

When the event occurs, a notification message is generated and sent as a reply to the asynchronous WS calls. Whenever a subscriber receives an event message, it will issue a new asynchronous WS call to continue receiving the event notifications.

Alternatively, callback endpoints can be provided that are able to asynchronously receive and process callback operation messages. When an event occurs, the eventing device sends eventing data to the subscriber's WS callback points.

4.2.3 WS + WS-Discovery

WS-* specifies WS-Discovery as a WS extension, which can add SD to enable .NET WS clients to search and discover other devices and the TEAHA gateway. WS-Discovery is, compared to UPnP, a more congruent fit with WSs. However, the extension is not supported on the .NET CF.

4.2.4 WS + UPnP

As WS-Discovery is not supported on the .NET CF, UPnP may be used as an alternative protocol for supplying SD support. The design depends on UPnP for advertising the Registration WS on the TEAHA gateway.

By allowing TEAHA services to be accessed through WS-Security extended WSs, encrypted and authenticated service interaction is possible. Consequently, unauthorized users are unable to intercept information about, or access TEAHA services.

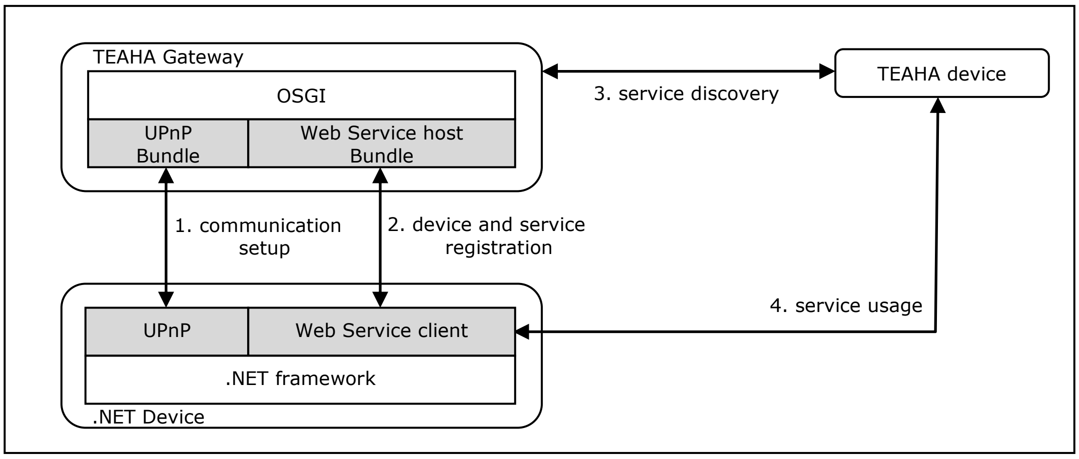
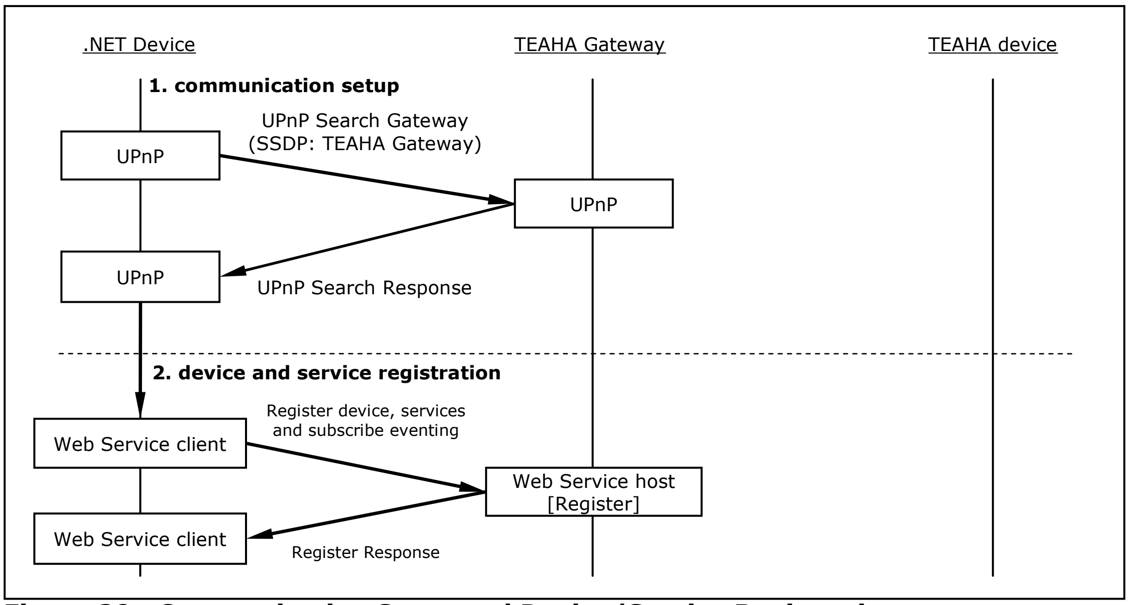


Figure 19: Mapping Web Services on TEAHA Services

During the communication setup, a .NET device broadcasts an unencrypted UPnP discovery message to locate the TEAHA gateway. The gateway then responds with information on how to access the Registration WS in a secure way. To protect against eavesdropping, the unencrypted UPnP messages do not contain any specific information about devices or services; any existing device is referred to as *generic device*.

Next, the .NET device registers itself and its services using the Registration WS on the TEAHA gateway. After which, the gateway updates the Service Registry, generates the corresponding .NET device proxy, and finally sends a registration confirmation back to the .NET device. Also, an asynchronous Eventing WS call is made to facilitate service requests to be issued by TEAHA devices.


.NET Device
TEAHA Gateway
TEAHA device
1. communication setup
UPnP Search Gateway (SSDP: TEAHA Gateway)
UPnP
UPnP
UPnP
UPnP Search Response
2. device and service registration
Register device, services and subscribe eventing
Web Service client
Web Service host [Register]
Web Service client
Register Response
Figure 20: Communication Setup and Device/Service Registration

When a .NET or TEAHA device searches for a particular device or service, the device correspondingly sends a WS or TEAHA search request to the TEAHA gateway. The gateway will then use the Service Registry to determine whether a device or service is available that satisfies the search request. If a successful match is found, the gateway will send a confirmation response back to the requesting device.

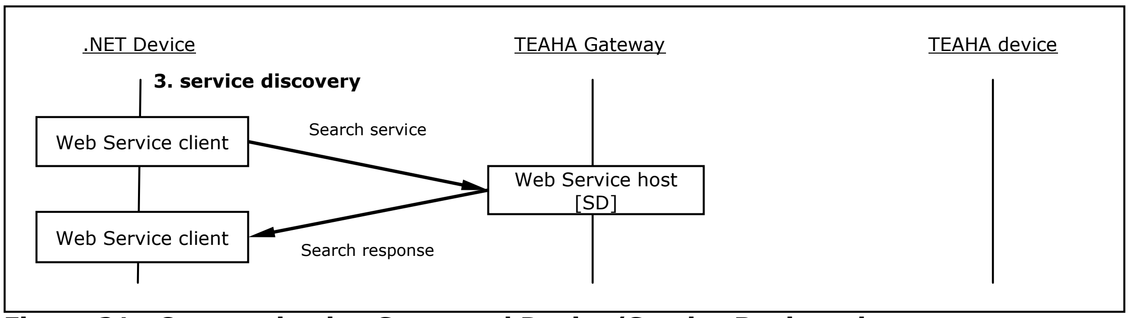


Figure 21: Communication Setup and Device/Service Registration

During service usage, if a .NET client accesses a TEAHA service, the client will send an asynchronous WS request to the TEAHA gateway. Following, the gateway translates the request into a TEAHA service call and forwards it to the TEAHA device that offers the service. Once the request has been dispatched by the TEAHA device, any service result or error is returned to the gateway. The gateway will then convert the result back into a WS response and send it to the .NET client as a reply to the initial asynchronous WS call.

In a very similar setting, if a TEAHA device wishes to access a .NET service, the device will send a TEAHA service request to the gateway. Next, the gateway converts the request and sends it as a response to an asynchronous Eventing WS call, which has been send to the gateway previously by the servicing .NET device. Following, the .NET device dispatches the request and any results or errors are included into a new asynchronous Eventing WS call that is send to the gateway. Finally, the gateway converts the response into a TEAHA result and forwards it to the initial requesting TEAHA device.

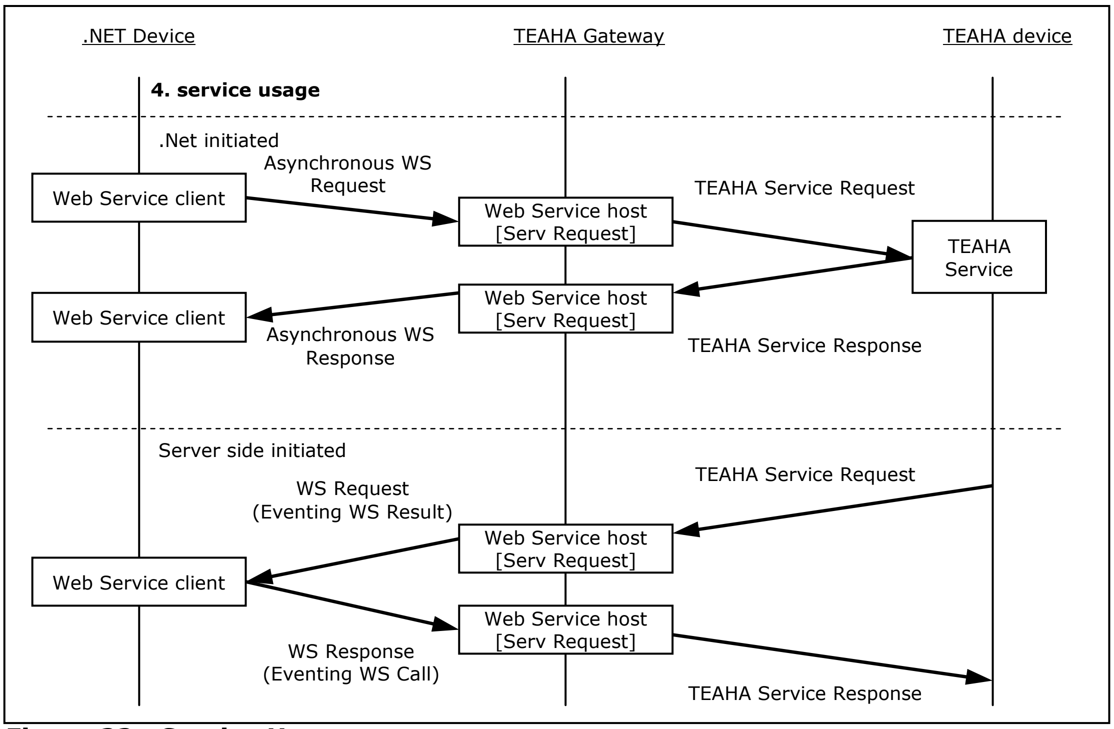


Figure 22: Service Usage

4.2.5 .NET Remoting + UPnP

.NET Remoting specifies a client-server model in which a *Remoting client* has access to a *Remotable Object* that is hosted on a *Remoting server*. Consequently, if the .NET client requires to access and host Remote Objects, it must be able to take the role of both Remoting client and server.

As stated before, *.NET Remoting* and other .NET communication technologies that have been reviewed in sections 3.7.6-3.11*,* do not offer SD. Hence, to facilitate the initial communication setup and to offer the required capability of searching and discovering devices and services, an additional SD protocol such as UPnP is necessary. Moreover, as .NET Remoting can use binary, SOAP or custom formatters, support for both communication formats may also need to be included.

The design uses a similar approach as the WS design, and relies on UPnP for .NET to search and discover the TEAHA gateway. After discovering the gateway, the .NET device registers itself and its services through a Remotable Object that handles registration. Moreover, the Remotable Object offers several other security and SD related services.

In addition, .NET Delegates and Events can be used with .NET Remoting to include eventing, and implemented according to a detailed tutorial available at [83].

OSGi support

In [84] a solution is given for accessing RMI/IIOP-based CORBA objects using .NET Remoting; it involves the mapping of Java types to IDL, and IDL to *Common Language Specification (CLS)* using a compiler. Moreover, in [85] an OSGi bundle is provided with a service that enables objects to be served remotely, by using CORBA *Dynamic Skeleton Interface (DSI)*. Combining the OSGi bundle with the aforementioned solution may provide a viable solution for adding .NET Remoting support to OSGi.

.NET support

.NET Remoting has been defined as an integral part of the .NET platform, however, support on the .NET CF is currently non-existent. Although, a solution is given for bridging .NET CF and .NET Remoting in [54], the provided solution does not enable real .NET Remoting support on the .NET CF, as it achieves bridging by adding additional extensions to the .NET Remoting server- and the .NET CF's WS-channels. Therefore, connecting a .NET CF device to a regular, non-extended .NET Remoting server is currently impossible.

Java Remoting Bridging

According to [86], the specification that governs .NET Remoting was being released to *Ecma International* as part of the CLI specification [56]. As a result, there are now several commercial products available that offer Java Remoting bridging. These Remoting engines generate proxies that allow .NET applications to invoke Java/J2EE systems, and Java/J2EE to invoke .NET applications [57].

As OSGi is based on Java, adapting commercial .NET Remoting bridging solutions into OSGi is presumably feasible. However, as OSGi is also destined for usage on resource-restricted devices, important factors such as memory, disk space and processor usage also need to be taken into concern.

Moreover, the design requirements do not allow usage of proprietary commercial standards or software.

Flexible Proxy Object Creation

.NET Remoting applications must include references to the Remotable Object Classes to compile successfully. Therefore, if a .NET device wishes to access a Remotable Object, the Remotable Object's Class is required by the .NET device in advance.

After compiling, .NET Remoting access is limited to object instances that are inherited from the Remotable Object Class. As the Remoting client must be able to access different devices during runtime, access that is restricted to a specific device may be problematic. Therefore, a dynamic way of generating Remotable Objects is highly recommended.

A solution can be offered by using dynamic proxy objects, which are proxy objects that can be created during runtime. A requirement for using dynamic proxy objects is that each proxy class must implement a particular interface, which has to be known in advance.

For creating dynamic proxies in Java, the *Code Generation Library (CGLIB)* [87] or the Java Dynamic Proxy API [88] can be used, while the .NET Framework can make use DynamicProxy.NET [89]; support for .NET CF is however not available.

4.2.6 Sockets + UPnP

Sockets are low-level communication methods that can enable communication between .NET and TEAHA. Besides sockets, which facilitate the actual transfer of data, UPnP is used for providing SD for discovering the TEAHA gateway and exchanging parameters for setting up sockets between endpoints.

Furthermore, additional security measures and proper conversion of data types and values between .NET and TEAHA will be required.

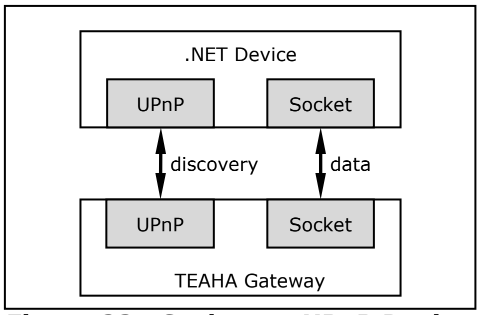


Figure 23: Sockets + UPnP Design

Data types and values within .NET are expected to be byte-incompatible within Java. Previous proposed design concepts include communication methods that are based on SOAP/XML, which enables interoperability between different OS's and programming languages. SOAP formatters, which are part of the Java API or available through external .NET code, can be applied to include SOAP communication [90].

In order for .NET devices and services to be discovered by other TEAHA devices, .NET devices and services need to be registered within the TEAHA Service Registry. Information regarding .NET devices and services will be securely transferred to the gateway using secure sockets. As sockets do not provide secure communication; additional security measures are required to offer authenticated and encrypted communication.

Moreover, a design solution that is based on sockets and UPnP is somewhat questionable. Seeing that UPnP can already solely provide SD, service eventing, and interoperability between .NET and TEAHA devices, it may suffice to only use UPnP for enabling communication.

4.3 Design selection

Several design concepts have been introduced and discussed comprehensively in the previous section. While considering the requirements that have been defined earlier in chapter 2, one of these concepts is eventually selected as the most suitable solution for composing the final design:

[R1] The design must allow .NET devices to transparently access and discover TEAHA services and devices.

[R2] The design must allow TEAHA devices to transparently access and discover .NET services and devices.

[R3] The design must include support for .NET CF.

[R4] The design must allow enforcement of policies on service access and discovery.

[R5] The design must support action and event driven user-service interaction.

[R6] The design must not be based or use proprietary standards and software.

[R7] The design preferably uses protocols that are well supported by the .NET and THEAHA framework; natively or by means of external software components.

[R8] The design must be scalable in order to support expansion of the number of devices for simultaneously accessing, discovering and offering services.

4.3.1 Reviewing

To select a final design, the design concepts in section 4.2 will be reviewed against the design requirements.

Web Services

Two designs based on WSs are introduced that allow TEAHA and .NET devices to access and discover each other's services. The designs only differ in the protocol being used for the initial communication setup. While the WS design using WS-Discovery is a more congruent design UPnP is, contrary to the former protocol, supported on the .NET CF.

WS communication is based on SOAP, which enables transparent access and interoperability between different OS's and programming languages **[R1] [R2]**.

Support for WSs is available on the .NET Framework; and with a few restrictions also on the .NET CF. While UPnP is fully supported using the Intel UPnP stack, WS-Discovery is not supported on the .NET CF **[R3]**.

WS-policy allows WSs to use XML for advertising their policies, and for WS consumers to specify their policy requirements. WS-Policy is not natively supported on the .NET CF, but can be included with help of external code [45]. Additionally, regular WS-Discovery and UPnP do not support the enforcement of policies on SD. However, as SD requests are handled by the TEAHA gateway, a design can be composed that regulates and allows the enforcement of policies from a single central point **[R4]**.

Native WS-Eventing and WS hosting are not supported on the .NET CF. However, support for eventing and server initiated communication can still be included using external code [45], asynchronous WSs, dedicated eventing sockets or polling; which have been discussed in section 4.2.2. Polling is however regarded as an inefficient and unsuitable approach, as it leads to communication overhead, while dedicated eventing sockets require ports that are free and not blocked by firewalls **[R5]**.

WSs and UPnP are based on Internet-based standards, which are promoted by W3C and *Organization for the Advancement of Structured Information Standards (OASIS)* as open standards. Additionally, all of the related WS-* specifications, including WS-Discovery, are also defined by W3C or OASIS as open standards **[R6]**.

As mentioned earlier, WSs are natively supported on the .NET Framework, and also partly on the .NET CF. Available OSGi bundles can extend the TEAHA framework to enable objects automatically be available as WSs, or allow WSs to be imported into an OSGI framework. While WS-Discovery is only supported on the .NET Framework, UPnP is also supported on the .NET CF with help of the Intel UPnP Stack **[R7]**.

The design concept introduces a solution for enabling communication between .NET and TEAHA devices through the use of a central TEAHA gateway that is responsible for the enforcement of policies, creation of device proxies, and enabling of SD. Therefore, the number of devices that connect to and are serviced by the gateway is restricted. To handle and distribute the service load, usage of additional gateways is required. This approach will introduce several new problems that are less trivial to solve. Problems that will arise are for example the distribution and update of encryption keys, distributed enforcement of policies, timing problems caused by latency, forwarding requests to the proper gateway, and discovery of services that are located on a remote gateway **[R8]**.

UPnP

UPnP is supported by TEAHA as part of the OSGi specification, and implemented through available OSGi bundles, such as the Domoware UPnP Base Driver bundle. Furthermore, Intel has released a UPnP stack that enables UPnP support in .NET applications. By using UPnP, TEAHA and .NET devices are capable of transparently accessing and discovering each other's services **[R1] [R2]**.

The design also offers support for .NET CF, since implementations of the Intel UPnP stack are available for both .NET and .NET CF **[R3]**.

Additionally, regular UPnP does not support enforcement of policies on SD. However, as SD requests are handled by the TEAHA gateway, a design can be composed in which enforcement of policies is regulated by the gateway **[R4]**.

By design, UPnP implements support for action as well as event driven user-service interaction. UPnP devices are able to offer services, and allow other network devices to subscribe to particular service events **[R5]**.

Similar to WSs, UPnP is also based on Internet standards and defined as an open, non-proprietary standard. Furthermore, the Intel UPnP stack, which enables UPnP support in .NET, is distributed with a royalty free license to copy, modify and create derivative works of the source code **[R6]**.

As mentioned before, UPnP can be implemented with help of the available OSGi bundles and Intel UPnP stack that both have been discussed in section 4.2.1 **[R7]**.

Comparable to the WS design, within the UPnP design concept, the TEAHA gateway handles policy enforcement, and conversion of service requests and responses between UPnP and TEAHA. Despite UPnP being designed to be scalable, adding TEAHA and UPnP devices to the gateway increases service load, which eventually can only be processed by distributing the service load between multiple gateways **[R8]**.

.NET Remoting

Comparable to other design proposals, the .NET Remoting design uses UPnP to allow .NET clients to search and discover the TEAHA gateway. Moreover, the .NET devices and their services also register through a Registration Service offered by a Remotable Object.

Devices can search or access other devices and services by sending a TEAHA or .NET request to the gateway. The gateway processes the request and uses the Service Registry to determine whether the device or service is available **[R1] [R2]**.

The .NET CF does not support .NET Remoting, while available solutions require adding extensions to the .NET Remoting channels on the client and server-side **[R3]**.

Similar to other designs, enforcement of policies may be implemented on, and handled by the central TEAHA gateway **[R4]**.

Support for action and event driven communication can be included with use of .NET Delegates and Events **[R5]**.

In section 4.2.5, a non-proprietary solution [84] has been discussed that enables .NET Remoting applications to access CORBA objects, while the bundle available at [85] allows CORBA objects to be serviced from OSGi **[R6]**.

For bridging .NET Remoting and .NET CF, a solution has been introduced that adds extensions to the .NET Remoting server and WS client channels. However, the solution does not truly enable .NET Remoting support. As a result, to allow .NET CF access OSGi CORBA objects, a custom .NET Remoting implementation on the .NET CF is required **[R7]**.

Due to the necessity of the TEAHA gateway to convert TEAHA and .NET Remoting service requests and responses, the .NET Remoting design will face the same scalability problems as all other designs. Enabling service load processing by multiple interconnected gateways is crucial for achieving scalability **[R8]**.

Sockets

Sockets are considered to be a low-level communication technology that does not offer SD. Hence, the communication setup is handled by UPnP to allow .NET devices to discover the TEAHA gateway. Also, proper conversion of data-types is required for enabling transparent service access and discovery. For this purpose, existing SOAP formatters can be used to establish OS and programming language independent communication **[R1] [R2]**.

The .NET CF does not offer support for raw sockets, nevertheless regular sockets are still supported **[R3]**.

Enforcement of policy enforcement on service access and discovery is handled by the TEAHA gateway **[R4]**.

As sockets offer two-way communication, devices are able to send and receive data in both directions. As a result, adding support for both action and event driven user-service interaction is most likely to be feasible **[R5]**.

The sockets design requires UPnP and SOAP support to be included, by utilizing custom or external code, such as the royalty free Intel UPnP stack, OSGi UPnP bundle, and available SOAP formatters **[R6]**.

Regular sockets are well supported in Java and .NET, while UPnP and SOAP functionality is included with help of the aforementioned external code **[R7]**.

The scalability of the design is low, due to the central policy enforcement and processing of SOAP based communication, which is regarded to be processing-intensive. In addition, the distribution of the service load amongst multiple gateways, introduce problems that are less trivial to solve **[R8]**.

4.3.2 Concluding

While reviewing the design concepts and taking the design requirements into account, a suitable design is chosen.

Sockets

The Sockets design requires implementing basic functionality that is mostly already part of several alternative communication technologies. Important design requirements such as eventing, SD, programming language and OS independent communication, and transparent service access are already provided by UPnP. Therefore, the sockets design is considered to be the least suitable choice.

.NET Remoting

Contrary to UPnP and WSs, .NET Remoting is not supported by OSGi or the .NET CF. Although several solutions for adding .NET Remoting were introduced, they were commercial and only available for Java. Furthermore, the proposed solution for adding .NET Remoting to the .NET CF is only an intermediate solution; it does not truly implement or utilize .NET Remoting, and requires adding extensions to the .NET Remoting server and WS client channels. Moreover, the .NET Remoting design requires additional SD protocols to be included, rendering it as a less suitable design solution.

Web Services

This leaves the WS and UPnP design concepts as remaining solutions that bear several similarities. Both UPnP and WS communication depend on HTTP and SOAP, which allows communication to be programming language and OS independent. Additionally, while UPnP revolves around providing SD, WSs can be extended with WS-Discovery.

Although, WSs are very popular and extensible, some important functionality is still missing on the .NET CF; such as support for WS-Eventing and WS hosting. Moreover, alternative solutions, such as polling, cause communication and processing overhead. While, dedicated eventing sockets may lead to problems, due to the requirement for open free ports and a possible custom firewall setup.

Unless native support for WS-Eventing and WS-Discovery is included, WSs are considered as a less suitable and preferred communication technology.

UPnP

Contrary to WSs, UPnP is part of the OSGi specification and supported by means of available UPnP bundles. In addition, UPnP includes eventing by design and has support for service advertising; devices can be discovered on the network and services can be listed. As a result, standard UPnP already largely complies with the design requirements.

Conclusion

In conclusion, and in view of the alternatives, a final design based on UPnP can be regarded as the most suitable and preferred choice for enabling communication between .NET and TEAHA devices. In addition, .NET clients will be able to offer services to other network devices within non-TEAHA environments using regular UPnP communication.

5 Design

The previous section concluded with a design based on UPnP being the most suitable and preferred choice for enabling communication. Following, the UPnP design will be defined in more detail and several additional security concepts will be introduced to provide secure communication.

5.1 Implementing UPnP

As described in section 4.2.1, implementation of UPnP is eased due to the availability of the OSGi *Domoware UPnP base bundle* and Intel UPnP .NET stack. The latter is a royalty free UPnP protocol stack, which can operate on both the .NET Framework and conveniently also on the .NET CF.

Compared to other design alternatives, UPnP is a more complete and suitable specification as it already covers many of the design requirements.

5.2 Securing UPnP

To provide secure communication with UPnP between .NET and TEAHA, changes to the UPnP protocol are necessary.

Several solutions are available for securing UPnP, including the proposed UPnP V1 Security specification, which has been discussed in section 3.15.3. The specification enables secure communication by providing authentication, encryption, and authorization.

5.2.1 Secure SD architecture

The UPnP V1 Security specification is considered to be less secure, compared to the secure SD architecture [91] that is based on STS. The architecture uses a separate Security Module, consisting of software and hardware, for handling authentication, encryption, and policy enforcement.

Although, the UPnP V1 Security specification offers a more congruent fit with UPnP, TEAHA's policy enforcement may be too complex or specific to be properly handled. Moreover, the secure SD architecture has proven to be a working concept within TEAHA. A design based on the Security Module and STS is, compared to a design that uses UPnP V1 Security, more congruent with the proposed secure SD architecture and TEAHA.

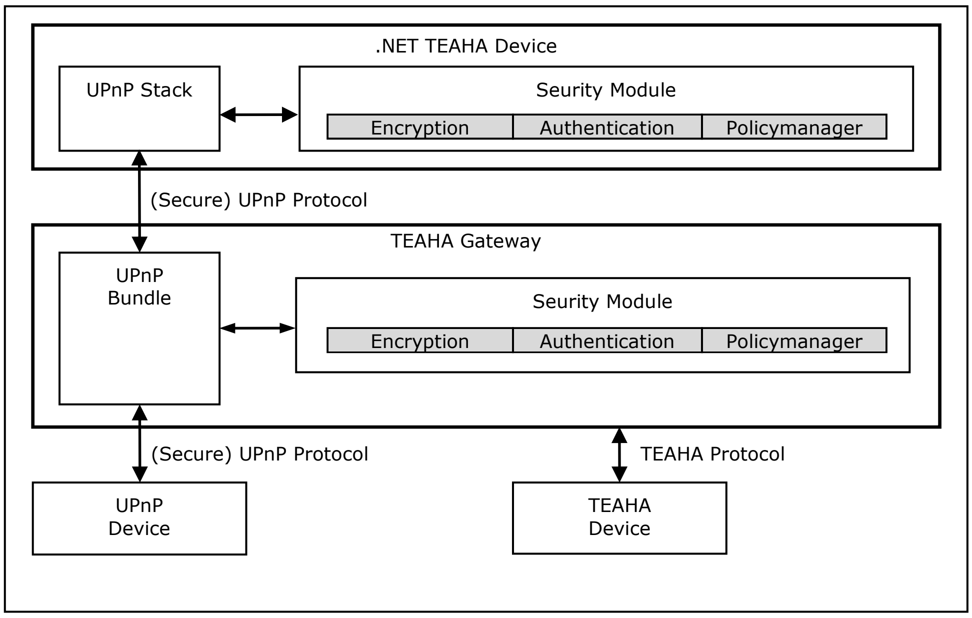


Figure 24: UPnP + Security Module Design

A clear separation between security and basic UPnP functionality, allows the original Intel UPnP stack and UPnP OSGi bundle to be used requiring minimal changes, and future updates and revisions of the UPnP stack and bundle to be adapted more easily.

Including the secure SD architecture into the UPnP design will require two different implementations of the Security Module, one suited for .NET and the other one suited for TEAHA. The following table shows an overview on the allowed communication security between Standard UPnP and Secure UPnP Stacks.

UPnP Stack	Standard	Secure
Standard	Not secure	Not secure/Unauthorized *
Secure	Not secure/Unauthorized *	Secure

Table 4: UPnP Communication Security

* Depending on device settings, communication between standard and secure UPnP stacks may be prohibited or be permitted to allow regular unsecure UPnP communication.

5.2.2 Security Module

The Security Module, as described in [91], contains four submodules that each may independently be implemented in either software or hardware. However, contrary to software, hardware is considered less likely to be forged and therefore more suitable for implementing critical security components such as the secure storage and crypto engine.
Independent of the number of offered device services, each TEAHA device only requires a single Security Module.

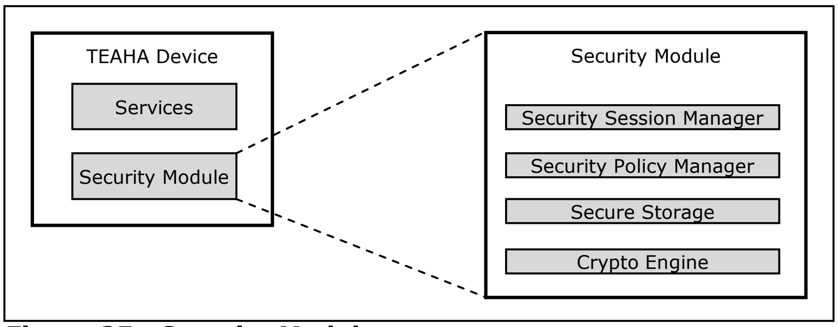


Figure 25: Security Module

Security Session Manager

The Security Session Manager module allows management of sessions and registration of session data. Each session may consist of a reference to an accessed remote service, or a set of references to network devices that accesses one of the services on the device.

Each session defines an encryption key that allows devices, to use authenticated and encrypted communication with a remote service or devices that access a local service provided by the device.

Security Policy Manager

The Security Policy Manager module is responsible for administrating and enforcing policies, and determines if a particular device has privileges to request a service action given a set of policy rules.

The policy manager may store a table that contains a device or service ID, a type that indicates whether the id refers to a service or device, required security mode, service action, default permission, and a list of devices that are granted or revoked to perform the service action.

Service ID (USN)	**Security Mode <auth., enc.>**	**Service Action**	**Default (allow/deny)**	**Allow List (UDN)**	**Deny List (UDN)**
SERV$_1$	<>	<discovery>	allow	[]	[D$_2$, D$_2$]
SERV$_1$	<auth>	<description>	deny	[D$_0$]	[]
SERV$_1$	<auth, enc>	<invoke>	deny	[D$_3$]	[]

Table 5: Service Policy Table

As an alternative to storing service permissions in a conventional table, a tree based approach, as discussed in [91], can be used to allow service permissions to be defined in a hierarchical and more logical manner. Each node within the permission tree represents a device or a service, and may access equal or lower level nodes.

In case a lower node requires access permission to a higher node, a permission pointer to the higher level node may be created at the lower node. The design will require multiple permission trees, each representing the service permissions for a particular service action.

Secure Storage

The Secure Storage module enables secure storage of critical data, such as private keys and user- and service-credentials. Hardware based secure storages are preferred above software based storages as they are more secure, tamperproof and less likely to be forged.

Crypto Engine

The Crypto Engine module is required for encrypting, decrypting, validating and signing *Protocol Data Units (PDUs)*. In order to perform decryption and encryption operations, the engine requires access to the private key stored within the secure storage. Hence, hardware based secure storages should be used with hardware based encryption engines.

As shown in [91], current smart-cards provide built-in secure storage and encryption functionality. They offer secure storage for credentials, and allow messages to be signed and encrypted on request. As both security information and encryption are internally processed within the smart-card hardware, exposure hazard of critical information is kept to a minimum.

Additionally, hardware based secure storage and crypto engines can relieve low-level processors from intensive encryption and decryption tasks.

5.2.3 Security modes

Three different security levels are defined, each applicable according to the required level of security.

Unsecure

Using the unsecure mode, standard UPnP protocol will be used; communication will not be encrypted or authenticated. UPnP devices will broadcast and send regular unencrypted and unauthenticated UPnP messages.

Authenticated

Authentication is the process of verifying the identity of an entity, such as services, clients or central directories. Security policies and access lists rely on the identity of entities for defining permissions on devices and services. Hence, when using authentication it is crucial that other devices or services cannot forge other entities.

Clients may have to authenticate in order to use a service, or servers may have to prove to clients that they are what they pretend to be. Options for achieving authentication include username and password methods, and stronger ones such as digital signatures. Once the identity of a device or service has been verified, authorization methods are used to determine the access and usage privileges of an entity.

Full Security

Enabling authentication and encryption offers full secure communication. Options for achieving authentication and encryption are STS and digital certificates. The latter include public keys that solve the problem, which adheres to symmetric encryption, of secure distribution of encryption keys. Public keys can be used for authentication, encryption and also for the secure exchange of symmetric session keys.

5.2.4 Replay Attacks

Standard unmodified UPnP PDUs do not contain timestamp or sequence information; if these were to be sent encrypted, the UPnP architecture would still be vulnerable to replay attacks. Hence, aside from adding a signature to protect message integrity, including a timestamp or sequence attribute, and caching is also a requirement to provide secure communication. In the event of a PDU being intercepted and reissued, the device offering the service will notice that the PDU is already cached and accordingly ignore the request.

Time Discrepancies

A common problem in the area of distributed computing is the time discrepancies that occur between the internal clocks of different devices. These discrepancies lead to unpredictable and inconsistent behavior; network devices may incorrectly deny PDUs with valid timestamps and remove them from cache too soon, or accept expired PDUs and keep them unnecessarily stored in cache.

Time Synchronization

Applications and communication that depend on timestamp information require time synchronization protocols such as *Network Time Protocol (NTP)* or *Simple Network Time Protocol (SNTP)* to minimize the effects of time discrepancies. The protocols have, depending on the network environment, an error margin of several milliseconds that needs to be considered with during the process of verification and caching.

Alternatively, if the allowed error margin is set large enough (a few seconds), network devices may simply use the gateway's timestamp to synchronize time. This approach does not require additional time synchronization protocols and may suffice for this design.

Sequence Attributes

Implementing sequence attributes will introduce several problems. Two separate devices or services could simultaneously send a PDU containing the same sequence number. Additionally, whenever a device is reinitiated, for example after a power-loss, the sequence number must continue from its last value. If the sequence number would simply be reset to zero or not restored correctly, expired PDUs could incorrectly contain valid sequence numbers and simply be reissued by a third party. Hence, sequence numbers require frequent expensive non-volatile write operations. Alternatively, less write operations are required if only future sequence values are stored instead, which do not get updated until the stored value is reached after a certain number of increments.

5.3 Centralized Policy Enforcement

The gateway is responsible for controlling policy enforcement, to allow policies to be conveniently managed from a central point, and consistency amongst policy rules easier to be attained.

If no custom policy rules have been defined for a device, the gateway will apply a general policy rule that is defined for all devices or for the particular device type. However, in case no policy rules have been defined by the gateway, the device should default to its own set of predefined policy rules. Moreover, device services for which custom policy rules have also not been defined are treated in a similar fashion.

5.4 Integrating STS

The STS protocol, as introduced in section 3.20, will be used for establishing common agreed keys for providing authentication and encrypting messages between two devices. During device initialization, a device will generate a secret random number that can be used for all future STS communication

Communication Procedures

In the following section a detailed overview is given of the events that take place during authentication, registration, and service requests. Each event has been defined as a device, followed by the action, and the unicast or multicast message it sends.

While full STS is used, encryption can easily be turned off, in case the required security mode only prescribes authenticated communication.

Furthermore, hashing can be used for referencing devices and previous issued PDUs to attain reduced PDU sizes. However, several hash methods have proven to be subject to collision attacks, as has been mentioned earlier in section 3.17.

Abbreviations

C_X	Public Certificate of D_X containing public key K_X and STS establishment parameters
D_X	Device X
D_{GW}	Gateway
$E_{X\text{-}Y}$	Encrypted by using common agreed key of two parties (D_X and D_Y)
K_X	Public Key of D_X
K_{SES}	Symmetric Session Key generated and provided by D_{GW}
Rx	Secret Random Number generated by D_X
S_X	Signed by D_X
$SERV_X$	Service X
X_X	g^{Rx} mod p

S_X adds a timestamp that marks the message only valid for a period of length, while the serial of C_X links to the signing device.

Assumptions

STS

- GF(p) is a finite field (p is a prime and public)
- g is a random public number (generator), $g \in GF(g)$

D_X

- has a digital certificate containing a matching K_X and private key pair
- uniquely identified and referenced by C_X
- uses a dedicated protocol, or timestamps received from D_{GW} to synchronize time
- able to sign, verify, decrypt and encrypt a PDU with a key
- able to generate a random number Rx

D_{GW} (in addition to the above assumptions)

- able to manage policy rules
- able to announce policy rule changes
- able to verify and authorize service request using policy rules

$SERV_X$

- is uniquely identified and referenced by its *Unique Service Name (USN)* and K_X

K_X

- (and therefore also the serial of C_X) uniquely identifies D_X

C_X

- Digital certificate containing public key, signed by a TEAHA CA
- Key establishment parameters are included within digital certificates.
- Key establishment parameters are included within digital certificates.

Advertisement

When a .NET device connects to the network, it will send a regular UPnP advertisement message extended with the modulus of its secret random number, required security mode, signature, and device certificate. The latter includes key establishment parameters and a public key which allows devices to verify the authenticity and integrity of messages.

Instead of broadcasting the certificate using unreliable HTTPMU, the certificate can be hosted separately while its URI location is included within the advertisement message.

Device D_A enters network

1. D_A Advertises on network $S_A(PDU_{SDDP})$, X_A, C_A (HTTPMU)

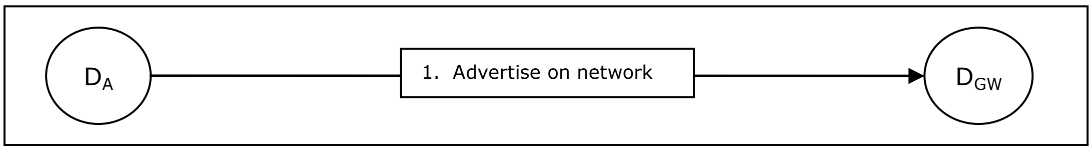


Figure 26: Device Advertising

Policy Checks
As policy rules are managed on the gateway, if an action is invoked on a servicing device, the device will send a policy check to the gateway. The gateway then determines if the action is granted or revoked, based on the policy rules. Finally, the check result and a reference to the check request are sent back to the device. In case no policy rules have been defined on the gateway, the device will apply its own set of default policy rules.

Registered device D_X checks policy with gateway D_{GW}

1.	D_X	Checks policy with D_{GW}	$S_B(E_{B\text{-}GW}(PDU_{CHECK}))$
2.	D_{GW}	Grant/Revoke action	$S_B(E_{B\text{-}GW}(PDU_{CHECKRESULT}\ ,\#PDU_{CHECK})$

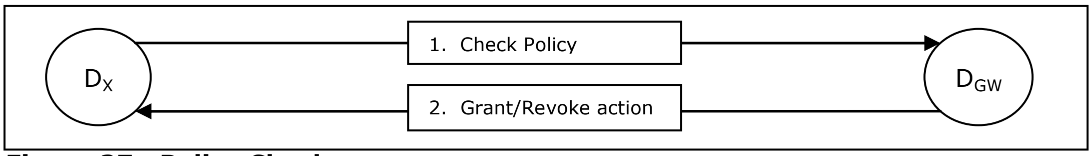


Figure 27: Policy Check

Authentication
After receiving the initial advertisement message of a device, the gateway uses the policy rules to determine whether to continue authentication and secure communication setup.

Following, the gateway uses the included key establishment parameters to calculate the modulus of its own secret random number. If the required security mode can be met by the network, both modulus numbers are concatenated, signed, encrypted, extended with the gateway's modulus number and certificate, and then sent back to the gateway. Similar to advertising, the certificate can be hosted separately and its URI location included within the message to avoid unreliable HTTPMU transfer.

Finally, the device concatenates both modulus numbers in a reverse order, then signs and encrypts the result, and sends it back to the gateway. After verification, both parties have a common agreed key for enabling authenticated and encrypted communication.

Device D_A sets up secure communication with gateway D_{GW}

1.	D_{GW}	Authenticates to D_A	$E_{A\text{-}GW}(S_{GW}(X_{GW}, X_A)), X_{GW}, C_{GW}$
2.	D_A	Authenticates to D_{GW}	$E_{A\text{-}GW}(S_A(X_A, X_{GW}))$

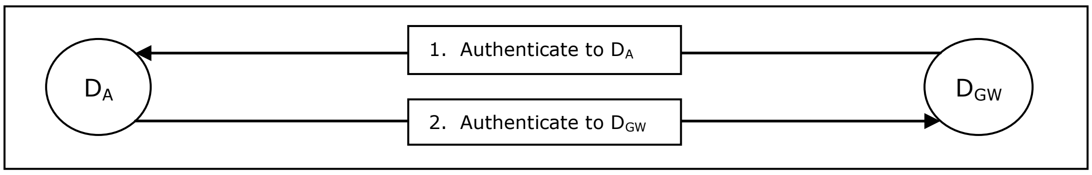


Figure 28: Device Authentication

Secure communication setup
In a similar way to devices that authenticate to the gateway, network devices can set up secure communication with other registered devices. The only real difference with the previous procedure is the policy check that can only be performed by the remote gateway.

Registered device D_X sets up secure communication with device D_A

1.	D_X	Authenticates to D_A	$E_{A\text{-}X}(S_X(X_X, X_A)), X_X, C_X$
	[D_A	Checks policy with D_{GW}]	
	[D_{GW}	Grants/revoke action]	
2.	D_A	Authenticates to D_X	$E_{A\text{-}X}(S_A(X_A, X_X))$

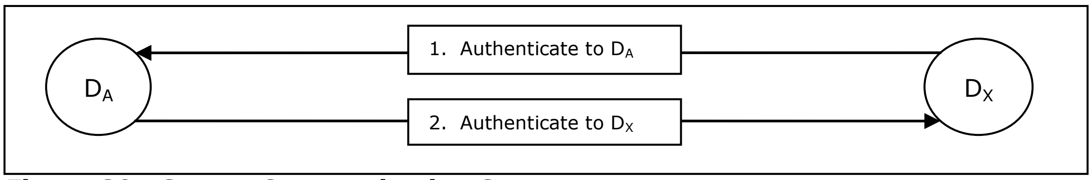


Figure 29: Secure Communication Setup

Registration
After secure communication has been established, the gateway commences device and service registration and requests device information from the device. The request is extended with a generated symmetric session key, which allows registered devices to encrypt messages that are broadcasted on the network. The gateway then encrypts, signs and finally sends the information request to the registering device.

Once the device receives the request, the device encrypts and signs the UPnP device description document and sends it back to the gateway.

Gateway D_{GW} registers device D_A and its services

1.	D_{GW}	Requests DevInfo D_A	$S_{GW}(E_{A\text{-}GW}(PDU_{DEVINFO}, K_{SES}))$
2.	D_A	Sends DevInfo to D_{GW}	$S_A(E_{A\text{-}GW}(PDU_{DEVINFO}\ D_A))$

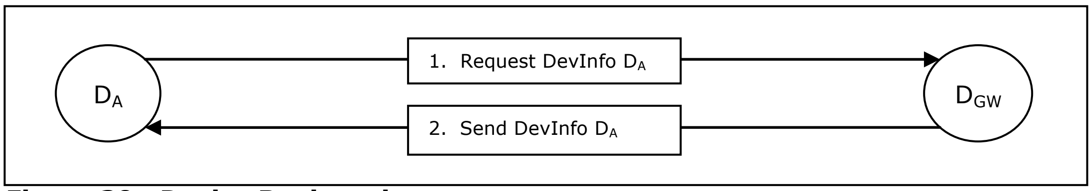


Figure 30: Device Registration

Device information request
In a similar way, to devices that registers to the gateway, network devices can request device information from other registered devices. The only differences with the previous procedure are the policy check, performed by the gateway, and the removed session key.

D_A registers device D_A and its services

1.	D_X	Requests DevInfo D_A	$S_X(E_{A\text{-}X}(PDU_{DEVINFO}))$
[D_A	Checks policy with D_{GW}]		
[D_{GW}	Grants/revoke action]		
2.	D_A	Sends DevInfo to D_X	$S_A(E_{A\text{-}X}(PDU_{DEVINFO}\ D_A))$

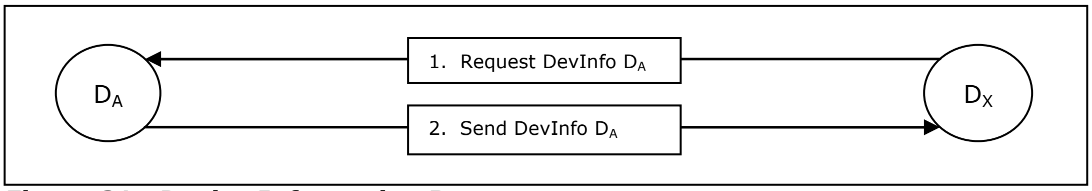


Figure 31: Device Information Request

Service Discovery
Once a device has been registered by the gateway, it may securely search other devices and services. The device first multicasts an encrypted and signed search request using correspondingly the received session key and its own private certificate.

Device D_A searches service $SERV_X$

1.	D_A	Searches service D_X	$S_A(E_{SES}(PDU_{SDDP}))$	(HTTPMU)
	[D_X	Checks policy with D_{GW}]		
	[D_{GW}	Grants/revoke action]		
2.	D_X	Notifies D_A	$S_A(E_{A\text{-}B}(PDU_{SDDP}))$	

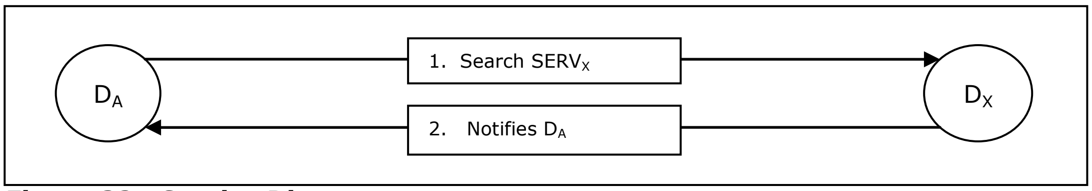


Figure 32: Service Discovery

However, as registered devices are able to decrypt the search request, despite the notification being encrypted, it may still reveal the service capabilities of the device.

To protect service capabilities, the search request could be unicasted directly to the gateway instead and encrypted using the agreed key. The gateway will then first execute a policy check and respond with the location of the searched service. As an added benefit, a separate policy check is avoided rendering it a more bandwidth-efficient approach.

Alternatively, devices may issue device information requests to all registered network devices. Although, this may also provide protection of service capabilities, on large networks this approach is a bit cumbersome. In addition, devices may not have abundant resources to store multiple device description documents.

Service Usage

Registered devices can encrypt and sign service requests and issue them to devices that provide the services. After the policy check grants the service request, the servicing device will dispatch the request and then encrypt, sign and eventually send the result back to the requesting device.

Device D_A requests service $SERV_X$ on device D_X

1.	D_A	Sends request to D_X	$S_A(E_{A\text{-}X}(PDU_{SOAP}))$
	[D_X	Checks policy with D_{GW}]	
	[D_{GW}	Grants/revoke action]	
2.	D_X	Sends result to D_A	$S_X(E_{A\text{-}X}(PDU_{SOAP}))$

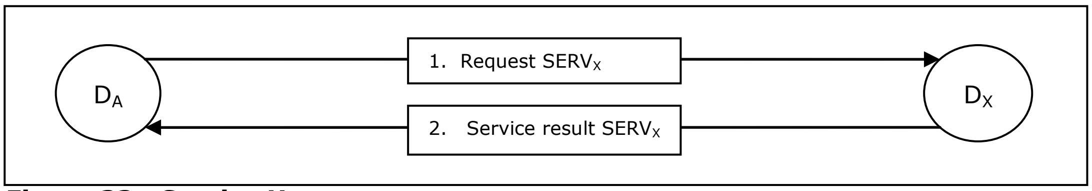


Figure 33: Service Usage

Service Subscription

Depending on the security requirements and policy rules, each service may need an individual symmetric service key.

After the subscribing device has sent the subscription request to the event source, a policy check is once again issued and handled by the gateway. If the gateway has granted the service subscription, the event source will send the service key to the subscriber. Service notifications can then onwards be encrypted and only be decrypted by the subscribed devices that have received the symmetric service key.

In addition to the above, a symmetric key must also be revoked in case a subscriber is banned from using a particular service.

Device D_A subscribes to service $SERV_X$ on device D_X

1.	D_A	Subscribes to $SERV_X$	$S_A(E_{A\text{-}X}(PDU_{GENA}))$	
	[D_X	Checks policy with D_{GW}]		
	[D_{GW}	Grants/revoke action]		
2.	D_X	Sends result to D_A	$S_X(E_{A\text{-}X}(PDU_{GENA}, K_{SERVX}))$	
3.	D_X	Broadcasts event notification	$S_X(E_{SERVX}(PDU_{GENA}))$	(HTTPMU)

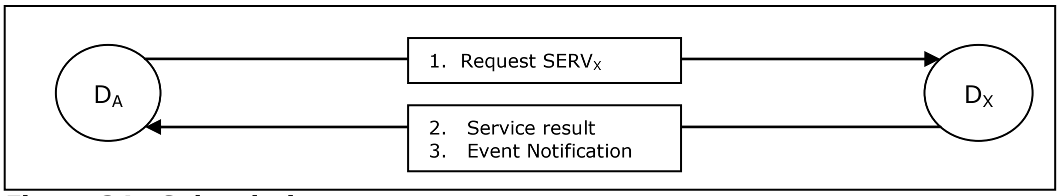


Figure 34: Subscription

Policy Update
When policy rules are updated, the gateway sends the policy updates to the servicing device for which the policy rules have changed. After receiving the policy update, the device can determine if any necessary actions are to be taken.

Whenever a subscriber is banned from a particular service, the service provider must revoke the service key and notify all subscribers of that service. First, using the old service key, the subscribers will receive a secure multicast notification message that includes a list of devices that are being banned. After which, the service provide will generate a new service key and redistributed the key within unicast messages.

Gateway D_{GW} broadcasts a policy update

1. D_{GW} Sends policy update to D_X $S_A(E_{GW\text{-}X}(PDU_{POLICY}))$
2. D_X Sends ban notification $S_A(E_{OLDSERVX}\ (PDU_{BANNED}))$ (HHTPMU)
3. D_X Sends new key to subscribers $S_X(E_{C\text{-}X}(PDU_{NEWKEY}, K_{NEWSERVX}))$

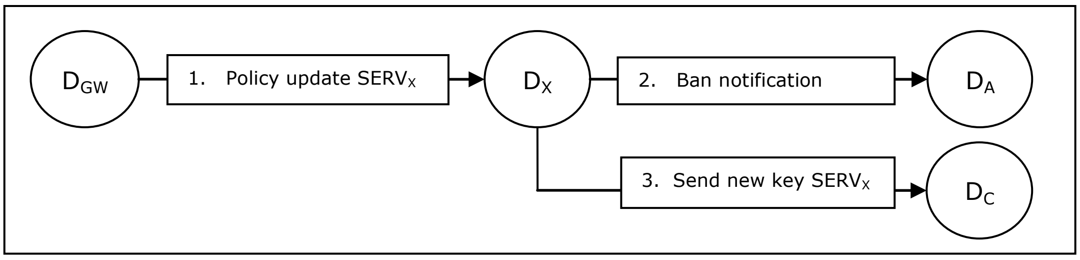


Figure 35: Policy Update

Alternatively, instead of sending the new service key to subscribers using multiple unicast messages, the service key could also be encrypted using the STS agreed keys of subscribers to create multiple individual encrypted instances of the key. Each version instance is then linked to the corresponding subscriber and added to the update message.

After the multicast message has been received, a subscriber will search for its own encrypted instance and decrypt it with the STS key. This approach may however introduce problems due to the unreliable nature of HTTPMU broadcasts.

Composed Services
Services that impose a serious security problem are device services that encapsulate or are composed of remote services. Without taking appropriate safety precautions, these services could allow devices to evade policy enforcement.

For example, device DA has valid access rights for service SERVB, but is prohibited from accessing SERVC. However, as SERVB is also composed of a remote call to SERVC, DA should consequently also be prohibited from accessing SERVA.

Since SERVC is only requested by DB, which has valid access permissions, DC has no reason to deny the service request. As a result, this will leave DB responsible for denying the service request issued by DA.

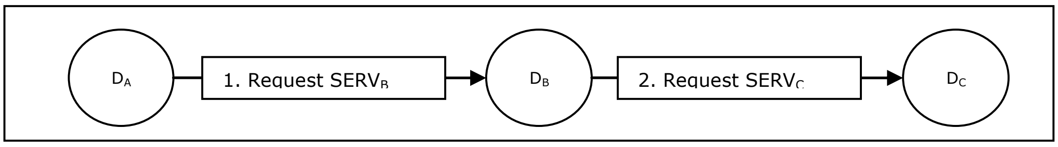


Figure 36: Composed Service $SERV_C$

Similar to the OSGi permission model, described in section 3.1.3, a solution can be offered by including a stack. The call stack consists of a sequence of calling device identities and denotes the forwarded path of the current service request. Upon DC receiving the request, it will notice DB being listed as a caller and consequently deny the operation.

Alternatively, during registration, device information can be extended and sent to the server to fully inform about any composed services. As a result, the gateway will be able to determine that DA has no valid access rights for accessing the composed service SERVB.

5.5 Concluding

In the previous section a design has been introduced that provides secure interaction between .NET and TEAHA devices. The design is based on the UPnP SD protocol, the concept of Security Modules and an authenticated version of the DH key exchange protocol (STS) to enable secure communication. The latter adds authentication to DH with help of certificates and digital signatures.

In addition, timestamps are included within UPnP messages to protect against replay attacks. As internal clocks of distributed devices generally cope with time discrepancies, time synchronization is considered mandatory when relying on timestamps.

Moreover, the design facilitates policy updates and policy enforcement that can be centrally managed by the TEAHA gateway. While, the policy update procedure allows symmetric keys to be updated and redistributed to subscribers in a secure manner.

6 Implementation

This chapter describes the prototype implementation of the final design introduced in the previous chapter. The general context overview diagram below describes how TEAHA devices communicate in a secure way, and how internal components control data and interact with other device components.

Unfortunately, the Intel does not provide a class diagram or a clear structure description of the Intel UPnP stack. However, contrary to the Intel stack, a sequence diagram is available as a future reference for the Domoware UPnP base driver [81].

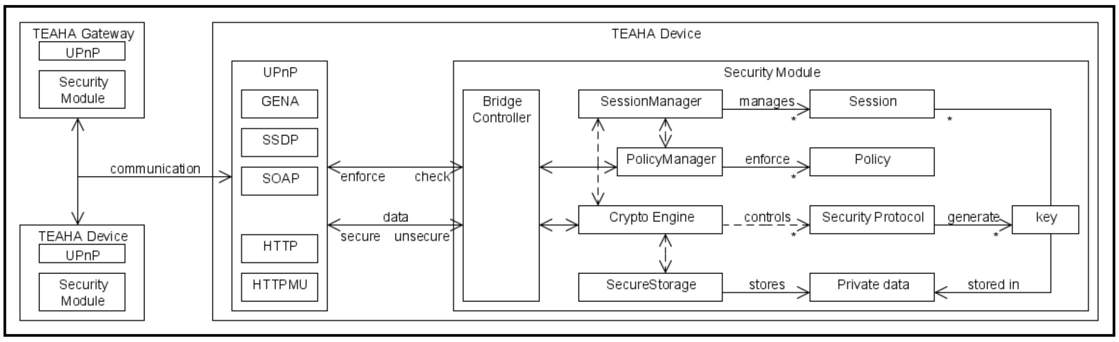


Figure 37: Overview Secure UPnP

6.1 Approach

Seeing that the final design consists of a .NET and OSGi implementation of the SM, the prototype implementation will be divided into two corresponding parts.

Guided by the context diagram, several class diagrams are first composed that will primarily relate to the SM. Subsequently, the diagrams are used for generating pseudocode, which abstracts and generalizes code in order to avoid being locked into code structures that are specific to a particular programming language. The generated pseudocode will therefore define the general code structure of both .NET and OSGi SM implementations.

Following, the code and data types are transformed into a custom .NET and OSGi prototype implementation. Because, the Intel UPnP Stack is written in C#, the .NET SM implementation will also be coherently done in C#.

Although, the SM is preferably realized in hardware, as data is securely stored and authentication and encryption may be solely handled by security chips, for practical reasons a software implementation is chosen. Nevertheless, converting the solution into a hardware based implementation is most likely to be feasible and eased due to the Crypto Engine and Secure Storage can be replaced by.

6.2 Class diagrams

The modules that compose the *Security Module (SM)*, introduced in section 5.2.2, and the data structures they operate on, are translated into separate class diagrams. The resulting diagrams are described in the following section.

In addition, a Bridge Controller is introduced that initiates and interlinks the submodules with help of a general *init* method that is defined by each submodule. Moreover, the Bridge Controller provides a set of service hooks that connects the SM with the Intel UPnP Stack and OSGi UPnP Base Driver.

6.2.1 Session Management

The Security Session Manager allows management of sessions, which are instances of the Session structure. Moreover, each session also references a unique key that is described as an instance of the Key structure.

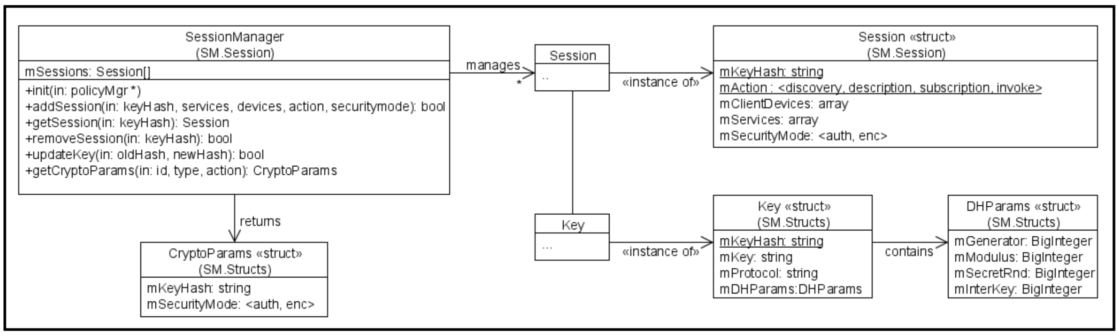


Figure 38: Session Management

Sessions

A session is defined as a communication agreement between several devices or services for a particular set of service actions. In addition, each session references to a single encryption key that is used for secure communication in the set of devices and services.

The Session structure defines two arrays of devices and service IDs, correspondingly *mDevices* and *mServices*, which may contain remote and/or own device and service references. The *mDevices* array will therefore consist of the serials of certificates that uniquely reference devices.

Furthermore, a session also defines a set of service actions, and security mode values, indicating if authenticated and/or encrypted communication is applied.

If only device references have been declared within a session, it will imply that the enlisted devices communicate and interact with each other and each other's embedded services using the referenced encryption key. This type of session declaration can be interpreted as a communication agreement of a default encryption key between the listed devices for a set of service actions.

However, in case service references have also been defined, the listed devices will use the referenced key for accessing the set of services. Therefore, the referenced key will be used for the particular set of services instead of any aforementioned default key.

In case a session does not contain any device references, it will imply that the list of services have not yet been requested or accessed by devices. The services are however already linked to a particular encryption key that is used for future secure communication.

Keys

Omitting actual key data and storing their references in sessions, allows confidential encryption keys to be separately stored in secure storage. Seeing that cryptographic processing is entirely handled by the Crypto Engine, if the key for a session is inquired by the Crypto Engine, the Session Manager will only need to return a reference to the actual key. As a result, only the Crypto Engine module and Secure Storage manager will require and granted sole access to secure storage, which in turn benefits security.

Similar to a session instance, referenced encryption keys are also instances of a structure. The Key structure defines a *mKeyHash* and *mProtocol* attribute. The first attribute references the key instance and consists of the *mKey's* computed SHA-1 hash, while the second one defines the key protocol used for generating the key.

In addition, if a session references a DH key, the *mDHParams* attribute will also include key establishment parameters, the generated secret random number *mSecretRnd*, and its public modulus number *mInterKey*. The latter is provided to other devices for initiating secure communication in order to perform secure service actions.

Manager
The Session Manager declares several methods that are provided to the Crypto and Policy Manager.

The *getCryptoParams* method is invoked by the Crypto Manager to acquire all necessary data for performing cryptographic processing. Aside from the general *init* method, the remaining methods will be accessed by the Policy Manager.

The *addSession* method adds a new session to the Session Manager if no sessions have yet been declared for the given *keyHash* value. Moreover, sessions that are being added may not result in inconsistency with existing sessions.

6.2.2 Policies

The policy manager enforces and manages a set of instances of the Policy Rule structure, and provides a *checkPolicy* function that is exported on the TEAHA gateway as an UPnP service.

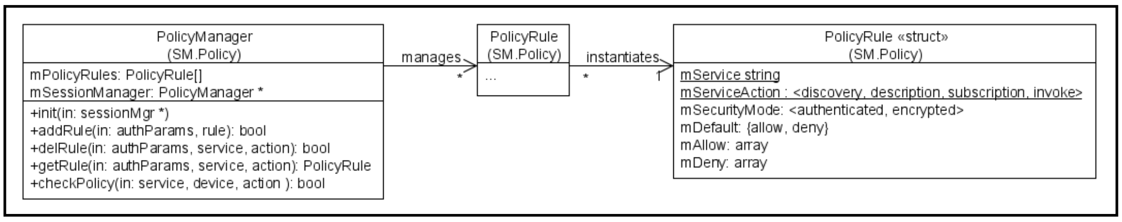

PolicyManager
(SM.Policy)
mPolicyRules: PolicyRule[]
mSessionManager: PolicyManager *
+init(in: sessionMgr *)
+addRule(in: authParams, rule): bool
+delRule(in: authParams, service, action): bool
+getRule(in: authParams, service, action): PolicyRule
+checkPolicy(in: service, device, action): bool
manages
*
PolicyRule
(SM.Policy)
...
instantiates
*
1
PolicyRule «struct»
(SM.Policy)
mService string
mServiceAction : <discovery, description, subscription, invoke>
mSecurityMode: <authenticated, encrypted>
mDefault: {allow, deny}
mAllow: array
mDeny: array

Figure 39: Policy Management

Policy Rules
The above Policy Rules are similar to the ones that were introduced in the policy table of section 5.2.2. Policy Rules are uniquely identified by the *tuple*: *mService* and *mServiceAction*, and contain a *mAllow* and *mDeny* array that include certificate serials of devices that are either allowed or denied to perform the stated service action. In case a device has not been listed in either one, the default policy action *mDefault* will be applied.

Moreover, policy rules also include a *mSecurityMode* attribute, which is more or less similar to the one defined within the Session structure. However, whereas the *mSecurityMode* attribute within a session defines the actual applied security mode, the attribute within a policy rule is to be interpreted as a minimum condition requirement for performing the set of service actions.

Therefore, the security mode defined in a policy rule may differ from the actual security mode declared in a session. For example, a policy rule may define an authenticated security mode, while the session could apply authenticated and encrypted communication.

Policy Manager
The Policy Manager provides several basic methods that are accessed by the Session Manager and exposed through the Intel UPnP stack and OSGi UPnP Base Driver.

The *checkPolicy* method, depicted earlier in section 5.3, is exported by the gateway as a UPnP service to offer centralized policy enforcement. Once a *checkPolicy* method call leads to a positive result, the Policy Manager will request the Session Manager to either include the device and service ID to an existing session, or to create a new one.

Similar to Sessions, adding a new Policy Rule must not lead to ambiguity with the existing set of policy rules.

6.2.3 Crypto

The Crypto Engine handles all cryptographic processing and initializes different encryption protocols. The Crypto Protocol base class consists of a list of cryptographic methods that is common to encryption protocols.

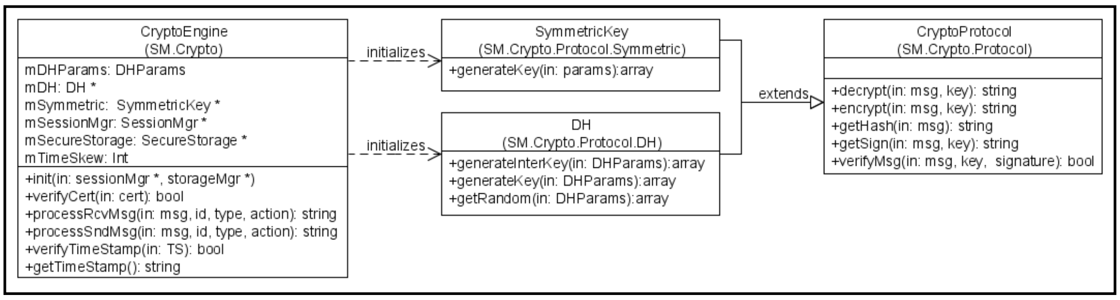

CryptoEngine
(SM.Crypto)
mDHParams: DHParams
mDH: DH *
mSymmetric: SymmetricKey *
mSessionMgr: SessionMgr *
mSecureStorage: SecureStorage *
mTimeSkew: Int
+init(in: sessionMgr *, storageMgr *)
+verifyCert(in: cert): bool
+processRcvMsg(in: msg, id, type, action): string
+processSndMsg(in: msg, id, type, action): string
+verifyTimeStamp(in: TS): bool
+getTimeStamp(): string
initializes
initializes
SymmetricKey
(SM.Crypto.Protocol.Symmetric)
+generateKey(in: params):array
DH
(SM.Crypto.Protocol.DH)
+generateInterKey(in: DHParams):array
+generateKey(in: DHParams):array
+getRandom(in: DHParams):array
extends
CryptoProtocol
(SM.Crypto.Protocol)
+decrypt(in: msg, key): string
+encrypt(in: msg, key): string
+getHash(in: msg): string
+getSign(in: msg, key): string
+verifyMsg(in: msg, key, signature): bool

Figure 40: Crypto Management

Crypto Protocols

The *DH* and the *SymmetricKey* classes represent encryption protocols that extend the *CryptoProtocol* base class. Both protocols define a *generateKey* method, which produces a key based on a set of key establishment parameters.

The list of common cryptographic methods contains a decrypt, encrypt, sign and verify procedure. The aforementioned methods all request a *msg* and *key* parameter, while the latter one defines an additional *signature* value that is verified against the other two input values.

The first two methods correspondingly receive an encrypted and an unencrypted message as input, and use the provided key value to process the message and subsequently return a decrypted and an encrypted result.

Finally, the last two complementary methods respectively return a message signature and a Boolean that indicates whether the message has a valid signature.

Crypto Engine

The Crypto Engine provides several methods that will be accessed by the Intel UPnP stack and Domoware UPnP Base Driver. The provided methods *verifyCert* and *processMsg* correspondingly verifies a certificate against a TEAHA Certificate Authority, and allows cryptographic processing of encrypted and unencrypted messages.

Both *processMsg* methods declare a *type* input parameter, which indicates whether the second *id* attribute refers to a device or a service. While the *processRcvMsg* method may need to decrypt incoming messages and verify their signature, the complementary *processSndMsg* may encrypt and sign the message depending on the applied Security Mode that is applied in the session.

In order to obtain the used encryption key, the *id* and *type* parameters are send to the Session Manager, which will then return a *mKeyHash* reference. The Crypto Engine can then proceed requesting the actual key data from the Secure Storage for further processing.

Additionally, a *verifyTimeStamp* method has been declared that takes an Unix UTC Timestamp string as input value, and compared against the current Unix UTC value of the local device's system time. Moreover, *mTimeSkew* contains a correction value that is applied during comparison, and is computed by subtracting the local timestamp from the ones provided by the gateway.

6.2.4 Secure Storage

The Secure storage Manager stores confidential information, such as the device's digital and TEAHA CA certificate, and instances of the key structure.

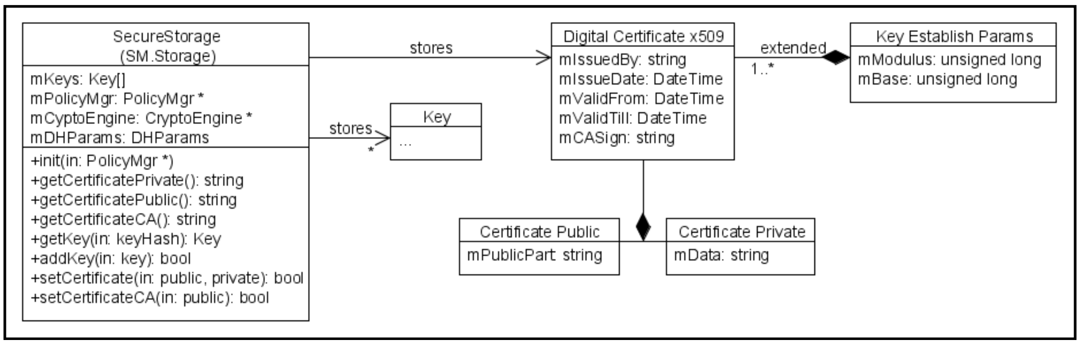


Figure 41: Storage Management

Certificates

Even though the digital certificates have been modeled as a class, in reality they will consist of local files that will contain the private or public parts of certificates.

The classes have only been added for the purpose of providing an overview on the internal data structures of certificates. As described by the diagram, DH key establishment parameters are included within digital certificate in order for receiving devices to initiate DH key agreement. However, the key establishment parameters may also be provided as predefined constant parameters that do not change over time. Moreover, the *mCASign* attribute represents the certificate signature generated by a TTP TEAHA Certificate Authority. Receiving devices can use this signature to verify the validity and authenticity of the certificate against the CA's public certificate.

Aside from storing digital certificates, the Secure Manager also manages instances of the *key* structure, which has been described previously in the Session Management class diagram. Also, as mentioned before, access to confidential data kept in secure storage is only granted to the Crypto Engine. Hence, the methods defined by the Secure Storage will solely be available to the Crypto Engine.

Besides methods for getting the public and private part of the device certificate, the Secure Storage manager also defines a *setCertificate* method, which references the public and complementary private part of a certificate. Concluding, for adding newly generated keys, the Crypto Engine provides the *addKey* method.

6.2.5 Bridge Controller

The SM Bridge Controller is responsible for initiating and interlinking the SM submodules. Moreover, the controller also bridges with the UPnP OSGi bundle or Intel UPnP Stack by providing service hooks to them.

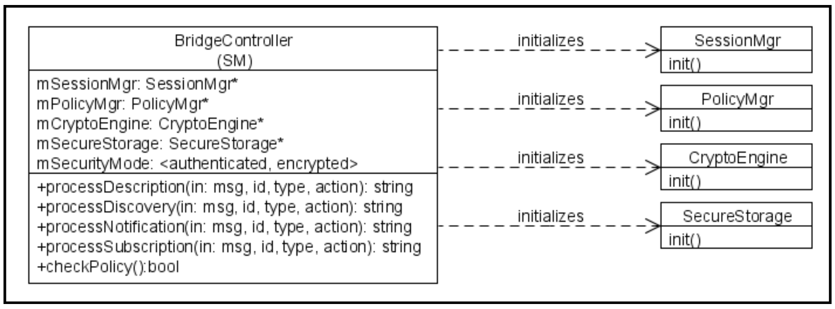

BridgeController
(SM)
mSessionMgr: SessionMgr*
mPolicyMgr: PolicyMgr*
mCryptoEngine: CryptoEngine*
mSecureStorage: SecureStorage*
mSecurityMode: <authenticated, encrypted>
+processDescription(in: msg, id, type, action): string
+processDiscovery(in: msg, id, type, action): string
+processNotification(in: msg, id, type, action): string
+processSubscription(in: msg, id, type, action): string
+checkPolicy():bool
initializes
initializes
initializes
initializes
SessionMgr
init()
PolicyMgr
init()
CryptoEngine
init()
SecureStorage
init()

Figure 42: Bridge Controller

The Bridge Controller contains several private members that each consist of a pointer to a SM submodule after initialization. In addition, the service hooks provided to UPnP are implemented in the classes that provide the corresponding services.

6.3 Extending UPnP Messages

This section describes how to facilitate secure communication by processing UPnP messages that are extended with additional security information. For reference purpose, several examples of regular UPnP messages are available in appendix 10.4.

To enable secure communication, an additional set of HTTP message headers are required:

<SIGN>	Denotes signature of the message.
<ID>	Contains the serial of the certificate referencing the sending device.
<KH>	Contains a keyhash reference to the key used for decrypting and verifying the message, which uniquely identifies a session.
<TS>	Contains a (time skew corrected) timestamp value, defined in Unix UTC.
<IK>	Contains device's DH interkey (public modulus of the secret random number).

When a message is send, the corresponding session is first determined in order to acquire the applied security mode. If the security mode does not set encryption or authentication, the message will be send unprocessed.

```
M-SEARCH * HTTP/1.1
ST: upnp:rootdevice
MX: 10
MAN: "ssdp:discover"
HOST: 239.255.255.250:1900
```

Figure 43: Regular UPnP Message

Authenticated

In case authentication is to be used, then a *TS* header is first added, after which the signature of the extended message is generated.

Following, the message is extended with the *SIGN* and *ID* headers, which correspondingly enclose the generated signature and the serial reference to the certificate for validating the signature.

Upon receiving the message, the receiver will first fetch the corresponding certificate that is referenced by the enclosed *ID* header. Next, the *TS* value is compared with the current Unix UTC timestamp on the local device to verify if the message is still valid.

After the timestamp has been validated, the receiver removes the *SIGN* and *ID* headers and verifies the remainder of the message against the *SIGN* value. In case the *TS* or *ID* headers have been tampered with, the signature verification will simply fail.

```
M-SEARCH * HTTP/1.1
ST: upnp:rootdevice
MX: 10
MAN: "ssdp:discover"
HOST: 239.255.255.250:1900
TS : 1057550400
ID: 2C41963FF4BC60B9451E2A8E66A24AE
SIGN: chIKzL59oxeJEnC2epDpX5pdnUY=
```

Figure 44: Authenticated UPnP Message

Encrypted

If encryption is also applied, then the message will first be entirely encrypted. Following, the *ID*, *TS*, *KH* and *SIGN* header are added after which the message is send.

Upon receiving the message, the *TS* value and the message signature are verified after which the encrypted message in the message body will be decrypted.

```
ID: 2C41963FF4BC60B9451E2A8E66A24AE1
TS : 1057550400
KH: S+P2CA0gKUtT5Gk/yGxg0hEhQgY=
SIGN: chIKzL59oxeJEnC2epDpX5pdnUY=
<DATA>  ZQAPAGkAEQAEAAQAZwACAA4AEQ……………….D0ASQAVABIAZAALABQARAAyACoANQB+
```

Figure 45: Full Secure UPnP Message

6.4 Code Structure

Based on the previous class diagrams, pseudocode can be created that eventually is fitted into a separate .NET and OSGi implementation. An overview of the created pseudocode, which depicts the general code structure, is provided in appendix 10.3.

The referenced code defines several object members, such as the *mSessions* member declared in *SessionManager*, which are similar to *Hashtables* in Java. However, C#, the matching equivalent of Java *Hashtables* are called *Dictionaries*.

```
Public Dictionary<string, Session> mSessions = new Dictionary<string, Session>();
```

Figure 46: C# Dictionary Type

The *Session* and *Key* structs are defined in Java as regular classes while the *struct* type is provided in C#.

```
public struct Key{
 public string mKeyHash;
 public byte[] mKey;
 public CryptoProtocol mProtocol;
 public DHParams mDHParams;
 }
```

Figure 47: C# Structs

Also Set variables such as *mAction* that may have multiple values declared, are defined in C# as *enum* with an additional *[flags]* attribute to allow bitwise operations. The declared enumeration are always powers of two, consequently, each bit position represents a single *enum* option.

```
[Flags]
public enum ServiceAction{
  discovery = 1,
  description = 2,
  subscription = 4,
  invoke = 8,
}

ServiceAction action = ServiceAction.discovery | ServiceAction.description;
```

Figure 48: C# Enum Sets

6.5 Implementation

This section discusses how to attach the service hooks provided by the SM within UPnP and how to implement DH and Certificates within .NET and OSGi.

6.5.1 Service Hooks

All service hooks provided by the Crypto Engine are attached, according to the OSI Seven Layer model, in the most upper layer: The Application Layer.

Upon receiving a secure message, the SM verifies and restores the original UPnP message by removing the additional security headers, and decrypting the data stored in the message body if encryption is used. Subsequently, the SM forwards the resulting UPnP message back to the UPnP stack or bundle for further UPnP processing and dispatching.

In the event of a UPnP message being send, the SM will use the included message headers and content to determine what parameters to apply for cryptographic processing.

Seeing that both receiving and sending messages require reading and writing header tags, the service hooks must be attached somewhere in the transport chain that allows message-level processing.

Consequently, service hooks that process received messages must be placed right after an incoming message has been converted from a lower-level transportable byte into its internal UPnP message representation. Whereas, service hooks that handle outgoing messages must be attached right before a message is serialized into its transportable byte representation.

Intel UPnP Stack

The service hooks, provided by the *Bridge Controller*, are primarily placed in the *UPnPDevice* class of the UPnP stack. The class offers methods for creating and managing UPnP Devices, and procedures for sending and processing received UPnP messages. Additionally, the service hooks must also be placed in the following classes: *UPnPService*, *UPnPControlPoint*, *UPnPComponent*, *HTTPRequest*, *HTTPSession* and *UPnPDeviceFactory*

OSGi UPnP Bundle

The OSGi bundle relies on the CyberLink UPnP Java library [92] for creating UPnP Devices, and handling sent and received UPnP messages. The aforementioned Service Hooks provided by the Bridge Controller are placed in the following UPnP Java Library classes: *HTTPSocket*, *HTTPMUSocket*, *HTTPUSocket*.

Furthermore, the *HTTPPacket* class is extended with a *removeHeader* procedure that facilitates the removal of the additional security header tags, which have been described in section 6.3.

6.5.2 Initializing Security Module

Upon creating a UPnP Device, the SM must be initialized and managed through the central Bridge Controller with a required Security Mode.

Within the Intel UPnP stack, the Bridge Controller is instantiated in the public constructer of the *UPnPDevice* class; whenever a UPnP Device is created, the SM will be initialized with the required security mode.

Furthermore, in order to properly close the application process and release memory, the *UPnPDevice.dispose()* method must be invoked upon exiting the application.

As for the OSGi UPnP bundle, the Bridge Controller is instantiated within the *BuildDevice* class of the aforementioned CyberLink UPnP Java Library.

6.5.3 DH

The authenticated version of DH (STS) is implemented in the .NET and OSGi framework and relies on certificates for authenticating devices. In order to use DH, suitable key establishment parameters (Generator and Prime) need to be selected. Instead of including these parameters within the certificates a set of predefined establishment parameters will be used for TEAHA devices.

.NET

Contrary to the full .NET Framework, .NET CF does not include the *ECDiffieHellmanCng* class to perform DH key exchange operations. As a result, sample code available at [93] have been converted and applied for including DH support. However, as the main purpose of the sample code was to demonstrate the basic concepts of DH, the resulting generated exchange keys were considered far too small to be suitably used for cryptographic processing.

Eventually, the package available at [94] had been used for providing DH. Initially, the package had been imported, compiled and accessed by the SM classes as a separate external module, which however lead to severe performance issues.

Nevertheless, by integrating the necessary classes (*BigInteger*) directly into the SM, and by removing unnecessary parts and performing code optimization; the average initialization time of the SM was successfully being reduced from fifty to merely eight seconds.

Java

Java provides support for DH through the additional *javax.crypto.KeyAgreement* and *javax.crypto.spec.DHParameterSpec* package. However, the key establishment parameters used in [94] have been declared as byte array values, whereas Java only accepts key establishment parameters that are defined as decimal values.

Nevertheless, [95] describes both decimal and corresponding HEX values of the generator and prime parameters as applied in [94].

6.5.4 Certificates

Certificates are essential to enabling authentication with DH (STS). RSA-SHA1 has been selected as the signing algorithm with certificates. Although 1024-bit asymmetric RSA keys are possible, according to [96] only RSA keys with a key size larger than 2048-bit are considered as to be secure.

.NET

In [97] has been described how to generate X.509 CA certificates and certificates that are signed by these CA certificates, while [98] describes how certificates can be installed on Windows Mobile-based devices. Moreover, both .NET Frameworks provide the *System.Cryptography.X509Certificates* namespace for processing X509 Certificates.

Within .NET, the private keys of certificates are stored within secure key containers that are referenced by a unique container name. In order to utilize a certificate's private key for creating signatures, the corresponding container name must be known. Unfortunately, contrary to the full .NET Framework, the .NET CF does not provide methods for retrieving the container name of a certificate.

As an alternative solution, the container name of the private key could be parsed from the *X509Certificate.ToString(true)* procedure that provides information about the a certificate's private key container. However, this turned out to be an unsuitable solution as the .NET CF implementation of the procedure does not include key container information.

Although, P/Invoke could be used to access the OS's underlying CryptoAPI library, this approach relies on specific Windows OS libraries that renders the solution only suitable for the Windows (Mobile) environment.

Fortunately, a Windows CE utility is available at [100] for importing private keys (PVK) files into a predefined custom container name. As the SM only needs to store one private key at a time (the certificate's private key of the device itself), the private key can always be referenced with the predefined container name.

In appendix 10.5 the commands are depicted that are used for creating the required TEAHA Root CA and TEAHA Device certificates that can be used with [100].

Java

In Java the corresponding certificate stores are referred to as Java Keystores (JKS), which are in essence password protected files that store keys and certificates. For the purpose of importing (PFX) certificates, which can contain both public and private keys, into Java Keystores, the procedure depicted in [101] can be used.

In addition, similar to .NET, Java provides the *java.security.cert.X509Certificate* packet for validating, verifying and processing X509 certificates.

7 Conclusions

Transparent and secure communication between .NET and TEAHA devices and services is feasible using the design introduced in chapter 5, which incorporates the Station-to-Station (STS) Protocol, UPnP and the concept of Security Modules (SMs).

Prior to composing the final design, several .NET communication protocols have been reviewed as potential solutions, including .NET Remoting, WSs and Sockets. The focus of attention has primarily been on the aforementioned technologies as they are the main communication protocols that are natively supported by the .NET platform. Although, many of these protocols seemingly offer viable solutions for implementing communication between .NET and TEAHA, during the design and implementation process the protocols introduced several unforeseen problems.

.Net Remoting Concept
Because .NET Remoting does not provide SD, a design concept had been introduced that combines .NET Remoting and UPnP. However, support on the .NET CF is nonexistent while available solutions do not enable true .NET Remoting.

Moreover, currently the OSGi framework does not provide support for .NET Remoting. Although, several commercial .NET Remoting bridging solutions are available for Java and .NET, in view of the project requirements and available alternatives, the protocol has been regarded as unsuitable for implementing communication.

Web Services Concept
WS support on the .NET CF is limited and only partially implemented; WS hosting is not supported, while the WS-Discovery, WS-Security and WS-Eventing specification have not been implemented. Nevertheless, with help of external code, the .NET CF can still be extended with WS-Security and WS-Eventing.

Seeing that support for WS-Discovery on the .NET CF is not available, a design has been introduced that has been based on both WSs and UPnP. The design requires the TEAHA gateway to host WSs, while .NET devices correspondingly take the role of WS clients. Although, the design is a viable and feasible solution, it was eventually regarded as a less suitable approach when compared to the design that is solely based on UPnP.

Sockets Concept
The Sockets design requires implementing basic functionality that is mostly already part of several alternative communication technologies. Important design requirements such as eventing, SD, programming language and OS independent communication, and transparent service access are already provided by UPnP. Therefore, the sockets design is considered to be the least appropriate choice.

Final Design
As TEAHA relies on UPnP for discovering, accessing and providing services, the UPnP protocol is considered as a congruent choice for enabling communication between .NET and TEAHA devices. Furthermore, the available Intel UPnP stack is conveniently suitable for both the .NET CF and full .NET Framework.

In addition, contrary to the other design proposals, the UPnP design is based on a single communication protocol, which results in a design that is comparatively less difficult to implement.

Security
In order to offer secure interaction between .NET and TEAHA, the concept of Security Modules (SMs) described in section 5.2.2, has been incorporated into the final design.

The SM concept defines a set of hard- and software based submodules that provide authentication, encryption and policy enforcement through service hooks. Furthermore, the SM is extended with an additional SM Bridging Controller module, whose purpose is to initialize and interlink the separate SM submodules, and to forward and attach the service hooks within the OSGi UPnP Base Driver and Intel UPnP Stack.

Moreover, if the SM is implemented in hardware, protection against cloning is provided while resource-restricted devices are relieved from intensive encryption and authentication processing.

Design Requirements
Seeing that many of the design requirements in section 2 relate to typical SD characteristics, a design that is based on the UPnP SD protocol will consequently largely comply with the aforementioned set of requirements:

[R1, R2] The design enables transparent access and discovery of .NET and TEAHA services with help of UPnP SD protocol.

[R3] The design provides support for the .NET CF, as the design relies on the Intel UPnP stack that can be used on the .NET CF as well as the full .NET Framework.

[R4] The design offers policy enforcement on service access and discovery through policy tables that are managed by the Policy Manager module.

[R5] The design supports action as well as event driven user-service interaction, which is a design requirement that is already a fundamental feature of the UPnP protocol specification. As a result, the UPnP design allows TEAHA and .NET devices to offer services, and enable other network devices to subscribe to specific service events.

[R6] The design is based on non-proprietary standards or software; UPnP is based on Internet standards and defined as an open standard, while all used external code are provided with licenses to copy, modify and create derivative works of the source code.

[R7] The design uses protocols that are well supported by the .NET and TEAHA framework; UPnP can be implemented with help of the available OSGi bundles and Intel UPnP stack.

[R8] The current proposed design is less suitable in terms of scalability. Within the UPnP design, the TEAHA gateway handles policy enforcement, and conversion of service requests and responses between UPnP and TEAHA. Consequently, increasing the number of connected TEAHA and UPnP devices will eventually result in a service load that can only be handled by multiple gateways or increasing current processing capabilities.

Concluding
The introduced design based on the UPnP SD protocol, SM concept and STS Key Exchange Protocol offers transparent and secure communication with policy enforcement between .NET and TEAHA devices and services. Furthermore, the design relies on open internet and non-propriety standards and software, and offers support for both action and event driven service interaction.

Seeing that policy enforcement, and conversion of service requests and responses is entirely handled by the central TEAHA gateway, the design is considered to be limited in terms of scalability. The increase of the number of connected TEAHA and UPnP devices will eventually require expanding the resource capabilities of the central TEAHA gateway or by distributing the service load amongst additional TEAHA gateways.

8 Recommendations for future work

This section discussed recommendations on future work and alternative designs using previous introduced communication technologies to improve the current design.

WCF
As has been often depicted, WS support is still limited on the .NET CF and therefore currently not suitable for implementation. However, as the .NET CF is still under major development and WCF support is slowly being added, it may eventually be advisable to reevaluate the possibility of using WCF instead.

WCF offers benefits over regular WSs in terms of performance and added security features. However, seeing that UPnP is being used in TEAHA, designs based on UPnP may still provide a more congruent fit with TEAHA.

Bandwidth-efficiency
In [99], compression of mobile WS interactions have proven to reduce message sizes and increase bandwidth-efficiency and WS performance when using slow and expensive connectivity, such as GPRS. At the expense of consuming CPU cycles at both the server and client, applying compression may be worth considering as both WSs and UPnP greatly rely on bandwidth-inefficient SOAP/XML based communication. Moreover, compressed and reduced message sizes also leads to less data that requires cryptographic processing, which also benefits overall performance.

UPnP Security
Although a UPnP design based on STS and the Security Module concept was chosen for implementing secure communication, a design based on UPnP Security V1 may offer a more congruent solution. Seeing that the specification enables the user to only encrypt parts within a SOAP message that are considered as confidential, a more efficient security solution is provided to the current one as it requires encryption of full-length messages. However, feasibility will largely depend on TEAHA's requirements for policy enforcement as they may be too complex to be properly defined and handled with UPnP Security V1.

Policy Enforcement
Several policy enforcement design proposals have been introduced, although feasible, the proposals are a bit cumbersome. A more communication-efficient design can be realized if policy rules are not solely stored and managed by the gateway, but also cached on devices that provide the services and enforce policies. This approach will however introduce several new problems such as managing cache renewal, available cache storage space, and acquiring consistency amongst policy rules distributed on several remote devices.

Encryption Protocols
The actual level of security provided depends largely on the encryption protocols being applied. Moreover, encryption protocols heavily influences performance and likely also PDU sizes. An in-depth comparison overview is required to select the most suitable protocol that offers an adequate level of security, acceptable performance requirements and that do not excessively decreases bandwidth-efficiency.

The actual level of security provided depends largely on the encryption protocols being applied. Moreover, encryption protocols heavily influences performance and likely also PDU sizes. An in-depth comparison overview is required to select the most suitable protocol that offers an adequate level of security, acceptable performance requirements and that do not excessively decreases bandwidth-efficiency.

Hardware Based Security Module
While the Security Module concept has only been implemented in software due to practical reasons, depending on the operational environment, a hardware based implementation may be more suitable. However, as this solution requires additional hardware, and unless vendors are willing to incorporate such facilities, providing a suitable implementation on existing mobile devices is less easy to achieve while guaranteeing device portability.

In addition, existing available security chips may already autonomously provide most of the required cryptographic processing, and relate to most of the recommendation topics above.

9 Bibliography

[1] TEAHA Consortium: TEAHA IST
http://www.teaha.org/

[2] Microsoft Corporation: .NET Framework Developer Center
http://msdn.microsoft.com/netframework/

[3] OSGi Alliance: OSGi - The Dynamic Module System for Java
http://www.osgi.org/

[4] Microsoft Corporation: .NET Remoting
http://msdn.microsoft.com/en-us/library/72x4h507.aspx

[5] Wikipedia: Service-Oriented Architecture (SOA)
http://en.wikipedia.org/wiki/Service-oriented_architecture

[6] OSGi Alliance: About the OSGi Service Platform - Technical Whitepaper (Rev 4.1), Jun 2007
http://www.osgi.org/documents/collateral/OSGiTechnicalWhitePaper.pdf

[7] OSGi Alliance: OSGi Technology
http://www.osgi.org/osgi_technology/

[8] OW2 Consortium: Oscar - OSGi framework
http://oscar.objectweb.org/

[9] The Knopflerfish Project: Knopflerfish - Open Source OSGi
http://www.knopflerfish.org/

[10] The Eclipse Foundation: Equinox
http://www.eclipse.org/equinox/

[11] The Eclipse Foundation: Eclipse - an open development platform
http://www.eclipse.org/

[12] Open Source Zone: Oscar
http://oszone.org/project/1210

[13] SourceForge: Oscar Bundle Repository
http://oscar-osgi.sourceforge.net/

[14] Sun Microsystems, Inc.: Java Management Extensions (JMX) Technology
http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/

[15] Wikipedia: .NET Framework
http://en.wikipedia.org/wiki/.NET_Framework

[16] Microsoft Corporation:
Differences between the .NET Compact Framework and the .NET Framework
http://msdn2.microsoft.com/en-us/library/2weec7k5.aspx

[17] Denise Barnes: Fundamentals of Microsoft .NET Compact Framework Development for the Microsoft .NET Framework Developer, Dec 2003
http://msdn2.microsoft.com/en-us/library/aa446549.aspx

[18] Dan Fox: The .NET Compact Framework, May 2003
http://www.samspublishing.com/articles/article.asp?p=31693

[19] .NET Compact Framework Team:
.NET Compact Framework version 2.0 Performance and Working Set FAQ, May 2005
http://blogs.msdn.com/netcfteam/archive/2005/05/04/414820.aspx

[20] Microsoft Corporation: Remoting
http://msdn2.microsoft.com/en-us/library/2weec7k5.aspx#Remoting

[21] Microsoft Corporation: Socket Programming
http://msdn2.microsoft.com/en-us/library/ms172494.aspx

[22] Microsoft Corporation: SocketOptionName Enumeration
http://msdn2.microsoft.com/en-us/library/system.net.sockets.socketoptionname.aspx

[23] Mark Prentince: Introduction to Windows Community Foundation for the .NET Compact Framework Messaging Stack, March 2007
http://blogs.msdn.com/markprenticems/archive/2007/03/27/introduction-to-windows-communication-foundation-for-the-net-compact-framework-messaging-stack.aspx

[24] Mono
http://www.mono-project.com/

[25] Richard Smith: DOT NET (or Mono) and Web Services, Jun 2004
http://www.novell.com/coolsolutions/feature/11227.html

[26] OpenNETCF Consulting, LLC
http://www.opennetcf.org/

[27] DotGNU Project: DotGNU Portable.NET
http://www.gnu.org/software/dotgnu/pnet.html

[28] Dave Marshall: Remote Procedure Calls (RPC), May 1999
http://www.cs.cf.ac.uk/Dave/C/node33.html

[29] OMG: CORBA
http://www.corba.org/

[30] Refsnes Data: SOAP Tutorial
http://www.w3schools.com/soap/

[31] Scripting News, Inc: XML-RPC Home page
http://www.xmlrpc.com/

[32] Wikipedia: XML-RPC
http://en.wikipedia.org/wiki/XML-RPC

[33] The Apache Software Foundation: Apache XML-RPC
http://ws.apache.org/xmlrpc/

[34] Oscar Bundle Repository: XML RPC
http://oscar-osgi.sourceforge.net/repo/xmlrpc/

[35] Charles Cook: XML-RPC.NET, Apr 2008
http://www.xml-rpc.net/

[36] Rick Strahl:
Creating and using Web Services with the .NET framework and Visual Studio.Net, Mar 2002
http://www.west-wind.com/presentations/dotnetwebservices/DotNetWebServices.asp

[37] Refsnes Data: Web Services Tutorial
http://www.w3schools.com/Web Services/default.asp

[38] Vangie Beal: Understanding Web Services, Oct 2005
http://www.webopedia.com/DidYouKnow/Computer_Science/2005/web_services.asp

[39] Wikipedia: List of Web Service specifications
http://en.wikipedia.org/wiki/List_of_Web_service_specifications

[40] Refsnes Data: WSDL Tutorial
http://www.w3schools.com/wsdl/

[41] Neil Cowburn: Consuming Web Services with the Microsoft .NET CF, Mar 2003
http://msdn2.microsoft.com/en-us/library/Aa446547.aspx

[42] Bea Systems: Using Polling as an Alternative to Callbacks, Version: 2006.0314.123656
http://edocs.bea.com/workshop/docs81/doc/en/core/index.html

[43] Reza Shafii: Creating Callback Enabled Clients for Asynchronous Web Services, Mar 2005
http://dev2dev.bea.com/pub/a/2005/03/callback_clients.html

[44] Don Box (Microsoft), Francisco Curbera (IBM) et al.: Web Services Eventing, Mar 2006
http://www.w3.org/Submission/2006/SUBM-WS-Eventing-20060315/

[45] Casey Chesnut: Compact Framework and WSE 2.0 Release, Jul 2004
http://www.mperfect.net/cfWse2/

[46] Wikipedia: WS-Policy
http://en.wikipedia.org/wiki/WS-Policy

[47] Roger L. Costello: Building Web Services the REST Way
http://www.xfront.com/REST-Web-Services.html

[48] Sun Microsystems, Inc.: Metro Web Services Technologies
http://java.sun.com/webservices/technologies/

[49] The Knopflerfish Project: The Knopflerfish Axis server
http://www.knopflerfish.org/repo/

[50] Microsoft Corporation: Web Services Enhancements 2.0
http://msdn.microsoft.com/en-us/library/aa894200.aspx

[51] Microsoft Corporation: Web Services Enhancements 3.0
http://msdn.microsoft.com/en-us/library/aa139619.aspx

[52] Microsoft Corporation: What's New in Web Services Enhancements (WSE) 3.0
http://msdn2.microsoft.com/en-us/library/ms977317.aspx

[53] Thiru Thangarathinam: NET Remoting Versus Web Services
http://www.developer.com/net/net/article.php/11087_2201701

[54] Bert Vanhooff, Davy Preuveneers, (K.U. Leuven, Department of Computer Science):
.NET Remoting and Web Services: A Lightweight Bridge between the .NET Compact and Full Framework, .NET Technologies 2005 Conference, April 2006
http://www.jot.fm/issues/issue_2006_04/article3.pdf

[55] Mark Strawmyer: .NET Remoting, Oct 2002
http://www.codeguru.com/csharp/csharp/cs_syntax/Remoting/article.php/c5871/

[56] Ecma International: Standard ECMA-335 - Common Language Infrastructure, Jun 2006
http://www.ecma-international.org/publications/standards/Ecma-335.htm

[57] Mohammad Adil Akif: J2EE and .NET interoperability through .NET Remoting, Dec 2005
http://blogs.msdn.com/mohammadakif/archive/2005/12/26/507416.aspx

[58] Payam Shodjai: Web Services and the Microsoft Platform, Jun 2006
http://msdn2.microsoft.com/En-US/library/aa480728.aspx

[59] Saurabh Gupta:Performance Comparison of Windows Communication Foundation (WCF) with existing distributed communication technologies, Feb 2007
http://msdn2.microsoft.com/en-us/library/bb310550.aspx

[60] Budi Kurniawan: Using .NET Sockets, Oct 2002
http://www.ondotnet.com/pub/a/dotnet/2002/10/21/sockets.htm

[61] Dean Stringer, Mike Vallabh: Summary - SOAP vs XML-RPC vs Others
http://webteam.waikato.ac.nz/Talks/WebServices/slide15-0.html

[62] Wikipedia: Jini
http://en.wikipedia.org/wiki/Jini

[63] The Apache Software Foundation: Apache River
http://incubator.apache.org/river/RIVER/index.html

[64] Robin Cover:
Microsoft Releases Web Services Dynamic Discovery Specification (WS-Discovery), Feb 2004
http://xml.coverpages.org/ni2004-02-17-b.html

[65] Tom Fout: Universal Plug and Play in Microsoft Windows XP, Aug 2001
http://technet.microsoft.com/en-us/library/bb457049.aspx

[66] Microsoft Corporation: Understanding Universal Plug & Play – whitepaper, Jun 2000
http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc

[67] Edward F. Steinfeld:
Home Entertainment Automation Using UPnP AV Architecture and Technology
http://www.go-embedded.com/UPnP White Paper.pdf

[68] UPnP Forum: UPnP Device Architecture 1.0, Jul 2006
http://www.upnp.org/specs/arch/UPnP-DeviceArchitecture-v1.0.pdf

[69] Wikipedia: Universal Plug and Play (UPnP)
http://en.wikipedia.org/wiki/Universal_Plug_and_Play

[70] Robin Cover:
UPnP Forum Releases New Security Specifications for Industry Review, Aug 2003
http://xml.coverpages.org/ni2003-08-22-a.html

[71] UPnP Forum: DeviceSecurity:1 Service Template, Nov 2003
http://www.upnp.org/standardizeddcps/documents/DeviceSecurity_1.0cc_001.pdf

[72] UPnP Forum: SecurityConsole:1 Service Template, Nov 2003
http://www.upnp.org/standardizeddcps/documents/SecurityConsole_1.0cc.pdf

[73] Intel Corporation: Intel® Tools for UPnP Technologies
http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/upnp/tools/index.htm

[74] Rich Salz, Securing Web Services, Jan 2003
http://webservices.xml.com/pub/a/ws/2003/01/15/ends.html

[75] Arjen Lenstra (Lucent Technologies), Benne de Weger (Technische Universiteit Eindhoven):
On the possibility of constructing meaningful hash collisions for public keys,
http://www.win.tue.nl/~bdeweger/CollidingCertificates/ddl-full.pdf

[76] Cryptography Research: Hash Collision Q&A, Feb 2005
http://www.cryptography.com/cnews/hash.html

[77] Martin Gudgin (Microsoft Corp), Marc Hadley (Sun Microsystems, Inc), Tony Rogers (Computer Associates International, Inc):
Web Services Addressing - SOAP Binding (W3C Recommendation), May 2006
http://www.w3.org/TR/2006/REC-ws-addr-soap-20060509/

[78] Ben Wright: Why can't I just use SSL to protect my Web Services?, May 2003
http://searchwebservices.techtarget.com/ateQuestionNResponse/0,289625,sid26_cid532568_tax294320,00.html

[79] Wikipedia: Proxy pattern
http://en.wikipedia.org/wiki/Proxy_pattern

[80] Microsoft Corporation: Interoperability Fundamentals, Dec 2003
http://msdn2.microsoft.com/en-us/library/ms978757.aspx

[81] DomoWare: Documentation – UPnP base driver
http://domoware.isti.cnr.it/documentation.html

[82] Christian Forsberg: Use Threading with Asynchronous Web Services in .Net Compact Framework to Improve User Experience, Feb 2004
http://msdn2.microsoft.com/en-us/library/aa446572.aspx

[83] Dmitry Belikov: .NET Remoting - Events. Events? Events!, Nov 2003
http://www.codeproject.com/csharp/RemotingAndEvents.asp

[84] ELCA: Accessing a RMI/IIOP-based CORBA object with .NET Remoting
http://iiop-net.sourceforge.net/rmiAdderDNClient.html

[85] Didier Donsez, CORBA Remote Service Impl (corbaservice.jar)
http://www-adele.imag.fr/users/Didier.Donsez/dev/osgi/corbaservice/readme.html

[86] Kenn Scribner: .NET/Java Interoperability: Apply the Proper Tool for the Job
http://www.developer.com/java/ent/article.php/10933_3351451_3

[87] cglib: Code Generation Library
http://cglib.sourceforge.net/

[88] Sun Microsystems, Inc.: Dynamic Proxy Classes
http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

[89] Jeppe Cramon: DynamicProxy.NET, Oct 2003
http://www.cramon.dk/dynamicproxy.htm

[90] Angelo Scotto: CompactFormatter -A generic formatter for the .NET Compact Framework
http://www.freewebs.com/compactFormatter/About.html

[91] Hans Scholten , Hylke van Dijk (University of Twente),
Danny De Cock, Bart Preneel (KU Leuven), Michel D'Hooge, Antonio Kung (Trialog):
Secure Service Discovery in Home Networks,
http://eprints.eemcs.utwente.nl/1649/01/ICCE2006-all.pdf

[92] Cyber Garage: CyberLink – Development Package for UPnP Devices for Java
http://www.cybergarage.org/net/upnp/java/index.html

[93] Lee Griffiths, Diffie-Hellman Key Exchange Example, Jan 2004
http://www.codeproject.com/KB/security/DiffieHellmanExample.aspx

[94] The Mentalis.org Team: DiffieHellman
http://www.mentalis.org/soft/class.qpx?id=15

[95] The Internet Society, H. Orman (University of Arizona):
RFC2414 - The OAKLEY Key Determination Protocol, Nov 1998
http://www.ietf.org/rfc/rfc2412.txt

[96] Wikipedia: Key size
http://en.wikipedia.org/wiki/Key_size

[97] Manu Cohen-Yashar: Creating X.509 Certificates using makecert.exe, Apr 2008
http://blogs.microsoft.co.il/blogs/applisec/archive/2008/04/08/creating-x-509-certificates-using-makecert-exe.aspx

[98] Microsoft Corporation: How to install root certificates on a Windows Mobile-based device
http://support.microsoft.com/kb/915840

[99] M. Tian, T. Voight, T. Naumowicz, H. Ritter, J. Schiller (Freie Universität Berlin):
Performance Considerations for Mobile Web Services
http://page.mi.fu-berlin.de/~tian/pdf/tian_et_al_aswn_elsevier_journal.pdf

[100] Jacco de Leeuw:
Personal Certificate Import Utility for Pocket PC 2003 and Windows Mobile, Jun 2007
http://www.jacco2.dds.nl/networking/crtimprt-org.html

[101] Chris Barber: Converting PFX Certificates to Java Keystores, Apr 2007
http://www.cb1inc.com/2007/04/30/converting-pfx-certificates-to-java-keystores

References in Appendix

[102] Payam Shodjai: Web Services and the Microsoft Platform, Jun 2006
http://msdn2.microsoft.com/En-US/library/aa480728.aspx#wsmsplat_topic32

[103] Henk Tiggelaar, Benchmark Results - Mono vs .Net, Apr 2007
http://readlist.com/lists/lists.ximian.com/mono-list/0/4079.html

[104] ToMMTi-Systems: Performance comparison C++, C# and Java
http://www.tommti-systems.de/main-Dateien/reviews/languages/benchmarks.html

[105] John Tertin: Mono vs. .NET Framework Benchmark Results
http://students.uwsp.edu/jtert146/monoframeworktest.xls

10 Appendix

The appendix includes a WS-* support and .NET performance comparison, several coding examples, Pseudocode and short descriptions on several types of UPnP PDUs.

10.1 WS-* Support in Microsoft Developer Platform

The following comparison overview is provided by [102].

Category	Protocol / Technology	ASMX 2.0	WSE 2.0	WSE 3.0	WCF	Windows Remote Mgmt (WinRM) on R2	WS for Devices (WSDAPI) on Vista
Core	Basic Profile 1.1	X	X	X	X		
	SOAP 1.1	X	X	X	X		
	SOAP 1.2	X		X	X	X	X
	WS-Addressing 1.0		X	X	X	X	X
Binary Data Transfer	DIME		X				
	MTOM			X	X		X
Other Transports & Encodings	TCP		X	X	X		
	UDP						X
	HTTP 1.1	X	X	X	X	X	X
	Text-XML	X	X	X	X	X	X
	Binary-Encoded XML				X		
	Binary Serialization		X	X			
Security	WS-Security 1.0		X	X	X		
	WS-Security 1.1			X	X		
	WS-SecureConversation 2005/02		X	X	X		
	WS-Trust 2005/02		X	X	X		
	Basic Security Profile 1.0		X	X	X		
	WS-Security SAML Token Profile 1.0 & 1.1				X		
Reliability	WS-ReliableMessaging 2005/02				X		
Transactions	WS-Coordination 2005/08				X		
	WS-AtomicTransaction 2005/08				X		
	WS-BusinessActivity						
Metadata, Policy & Binding	WS-MetadataExchange 2004/09				X		X
	WSDL 1.1	X	X	X	X	X	X
	WS-Policy 2006/03				X		
	WS-PolicyAttachment 2006/03				X		
	WS-SecurityPolicy 2005/07				X		
Management & Devices	WS-Management 1.0					X	
	WS-Transfer 2004/09					X	
	WS-Enumeration					X	
	WS-Eventing					X	X
	WS-Discovery						X
	Devices Profile						X

10.1 Performance comparison Mono/.NET

The table below shows the results [103] of a performance comparison between .NET and Mono, using C# benchmark code [104]; other comparisons are available at [105].

	Mono 1.2.3.1	*.NET 2.0*	*Percentage (%)*
Int arithmetic	*8094*	*5812*	*+ 39.26*
Double arithmetic	*12141*	*7249*	*+ 67.49*
long arithmetic	*26406*	*16265*	*+ 62.35*
Trig	*2749*	*2281*	*+ 20.52*
IO	*3204*	*2499*	*+ 28.21*
Array	*406*	*203*	*+ 100.00*
Exception	*3719*	*26687*	*- 617.59*
HashMap	*234*	*124*	*+ 88.71*
HashMaps	*5265*	*3999*	*+ 31.66*
HeapSort	*609*	*531*	*+ 14.69*
Vector	*9953*	*9890*	*+ 00.64*
Matrix Multiply	*71562*	*33687*	*+ 112.43*
Nested Loop	*10359*	*22265*	*- 114.93*
String Concat.(fixed)	*578*	*359*	*+ 61.00*
Total C# benchmark	*155279*	*131851*	*+ 17.77*

10.2 Usage example Intel UPnP stack

```
UPnPLightDevice = UPnPDevice.CreateRootDevice(200,1,"web\\");
UPnPLightDevice.HasPresentation = false;
UPnPLightDevice.FriendlyName = this.Text;
UPnPLightDevice.Manufacturer = "Intel Corporation";
UPnPLightDevice.ManufacturerURL = "http://www.Intel.com";
UPnPLightDevice.ModelName = "Intel CLR Emulated Light Bulb";
UPnPLightDevice.ModelDescription = "Software Emulated Light Bulb";
UPnPLightDevice.ModelURL = new Uri("http://www.Intel.com/xpc");
UPnPLightDevice.ModelNumber = "XPC-L1";
UPnPLightDevice.StandardDeviceType = "BinaryLight";
UPnPLightDevice.UniqueDeviceName = Intel.Utilities.Guid.NewGuid().ToString();

//LightService
UPnPLightService =
   new UPnPService(1,"SwitchPower.0001","SwitchPower",true,this);
UPnPLightService.AddMethod("SetTarget");
UPnPLightService.AddMethod("GetStatus");

//get command
UPnPStateVariable UPnPStatusVar =
   new UPnPStateVariable("Status",typeof(bool),true);
UPnPStatusVar.AddAssociation("GetStatus","ResultStatus");
UPnPStatusVar.Value = false;
UPnPLightService.AddStateVariable(UPnPStatusVar);

//set command
UPnPStateVariable UPnPTargetVar =
   new UPnPStateVariable("Target",typeof(bool),false);
UPnPTargetVar.AddAssociation("SetTarget","newTargetValue");
UPnPTargetVar.Value = false;
UPnPLightService.AddStateVariable(UPnPTargetVar);

UPnPLightDevice.AddService(UPnPLightService);


// Dimmable device
UPnPDimmerService =
   new UPnPService(1,"DimmingService.0001","DimmingService",true,this);
UPnPDimmerService.AddMethod("SetLoadLevelTarget");
UPnPDimmerService.AddMethod("GetLoadLevelStatus");

//get command
UPnPStateVariable UPnPLevelStatusVar =
   new UPnPStateVariable("LoadLevelStatus",typeof(byte),true);
UPnPLevelStatusVar.AddAssociation("GetLoadLevelStatus","RetLoadLevelStatus");
UPnPLevelStatusVar.Value = (byte)100;
UPnPLevelStatusVar.SetRange((byte)0,(byte)100,null);
UPnPDimmerService.AddStateVariable(UPnPLevelStatusVar);

//set command
UPnPStateVariable UPnPLevelTargetVar =
   new UPnPStateVariable("LoadLevelTarget",typeof(byte),false);
UPnPLevelTargetVar.AddAssociation("SetLoadLevelTarget","NewLoadLevelTarget");
UPnPLevelTargetVar.Value = (byte)100;
UPnPLevelTargetVar.SetRange((byte)0, (byte)100, null);
UPnPDimmerService.AddStateVariable(UPnPLevelTargetVar);

UPnPLightDevice.AddService(UPnPDimmerService);

UPnPLightDevice.StartDevice();
```

10.3 Pseudocode Implementation

```
Package sm.parameters
 /**
  * Contains a list of parameters that have predefined set of values
 **/
Type ServiceAction { discovery, description, subscription ,invoke }
Type SecurityMode { authentication, encryption }
Type DeviceService { device, service }
Type AllowDeny { deny, allow }
Type Protocol { dh, symmetric }


Package sm.structs

Import SM.Parameters

Struct Session {
   mKeyHash: string
   mDevices: array
   mServices: array
   mAction: ServiceAction
   mSecurityMode: SecurityMode
}

Struct Key {
   mKeyHash: string
   mKey: string
   mKeyProtocol: Protocol
   mParams:array
}

Struct PolicyRule {
   mID: string
   mType: DeviceService
   mAction: ServiceAction
   mSecurityMode: SecurityMode
   mDefault: AllowDeny
   mAllow: array
   mDeny: array
}

Struct CryptoParams {
   mKeyHash: string
   mSecurityMode: SecurityMode
}

Struct Certificate {
   publicPart: string
   privatePart: string
}


Package sm.session

Import sm.parameters
Import sm.structs

Public Static Class SessionMgr {
   Private Session[keyHash] mSessions

   /**
    * Accessed by PolicyManager for adding a session based on allowed policy rules
   **/
   Public bool addSession( string keyHash, string id, DeviceService type, ServiceAction action){
      * Check if keyHash exists in session pool, if not add the session to the pool *
       return boolean indicating if session was added
   }
   /**
    * Accessed by CryptoEngine for adding a session based on allowed policies
   **/
    Public CryptoParams getCryptoParams( string id, DeviceService type, ServiceAction action){
      * Perform search in session for service or device id, based on type. Return CryptoParams *
      return fetched CryptoParams (if available)
   }

   /**
    * Accessed by Accessed by CryptoEngine if a particular key is getting updated
   **/
   Public bool updateKey( string oldHash, string newHash){
      * Get session with oldHash reference and replace value with newHash *
       return boolean indicating if key was updated
   }
}


Package sm.policy

Import sm.constants
Import sm.structs

Public Static Class PolicyMgr {
   Private PolicyRule[] mPolicyRules

   /**
    * Forwarded as UPnP service for checking policy for a particular service, device and service action
   **/
   Public bool checkPolicy(string device, string service, ServiceAction action){
      * Search for rules with ServiceID and action as key and perform check and return result *
       return boolean indicating if action is allowed
   }

   /**
    * Forwarded as an UPnP service allowing authorized devices to manage policy rules
   **/
   Public bool addRule( array authParams, PolicyRule rule){
      * If authorized, add rule to mPolicyRules *
       return boolean indicating if rule is added
   }
   Public bool delRules( array authParams, string ServiceID){
      * If authorized, search for rules and remove from PolicyRule *
       return boolean indicating if rule is removed
   }
   Public PolicyRule[] getRule ( array authParams, string ServiceID){
      * If authorized, search for rules with ServiceID and return as result *
      return fetched rules (if available)
   }
}

Package sm.crypto

Import sm.crypto.protocol
Import sm.crypto.protocol.sts
Import sm.crypto.protocol.symmetric

Public Static Class CryptoEngine {
   Private mKeyParams
   Private static Protocol STS
   Private static Protocol Symmetric

  Public CryptoEngine(){
      STS = new STS()
      Symmetric = new Symmetric()
   }

   Public bool verifyCert(string cert, string certCA){
       return boolean indicating if certificate is valid
   }
```

```
     /**
    * Accessed by controller for processing messages based on input parameters
   **/
   Public string processMsg(string msg, string id, type type){
      SessionMgr->getCryptoParams()
       return string consisting of the processed message
   }
}

Package sm.crypto.protocol

Interface Protocol{
   Public string decrypt(string msg, string key)
   Public string encrypt(string msg, string key)
   Public string sign(string msg, string key)
   Public bool verify(string msg, string key, string signature)
   Public array generateKey(array params)
}


Package sm.crypto.Protocol.DH

Class STS implements Interface Crypto Protocol{
   Public string generateKey(array params){
      STS
      return string
   }
...
}


Package sm.crypto.protocol.symmetric

Class Symmetric implements Interface Crypto Protocol{
   Public string generateKey( array params){
      return array with the encryption key as a single element
   }
...
}


Package sm.storage

Class SecureStorage{
   Private array mStorage
   Private TeahaCertificate
   Private DeviceCertificate

   Public string getCertificatePrivate(){
      return string
   }

   Public string getCertificatePublic(){
      return string
   }

   Public string getKey( string keyHash){
       return string consisting of
   }

   Public bool addKey(string key){
       return boolean indicating if key is added
   }
}
```

10.4 UPnP Messages

Several examples of UPnP messages that are sent during UPnP communication are described in the following section. The listed message examples are sent by the UPnP light and control point application, which are included within the Intel UPnP stack package.

Most UPnP messages that start with a *NOTIFY* or *M-SEARCH* header are multicasted using HTTPMU, while other messages rely on unicast HTTP or HTTPU. Due to the unreliable nature of UDP, multicast messages are advice to be sent more than once.

A complete overview of the UPnP architecture and description of UPnP messages is provided by [68].

10.4.1 Notification

When a UPnP device enters the network it will multicast multiple messages, one for the root device and one for each service or embedded device it may contain.

The *Notification Sub Type (NTS)* *sddp:alive* indicates that the device is available on the network. The *Unique Service Name (USN)* uniquely identifies the device or service and consists of a *Unique Device Name (UDN)* and its *Notification Type (NT)*.

In addition, the device is required to resend a notification message within the max-age expiration value, defined in seconds, to notify that the device or a service is still available.

```
NOTIFY * HTTP/1.1
NT: upnp:rootdevice
USN: uuid:cbc7208e-b395-4e61-acd0-9883071ef021::upnp:rootdevice
NTS: ssdp:alive
SERVER: Windows NT/5.0, UPnP/1.0, Intel CLR SDK/1.0
LOCATION: http://10.0.1.1:61342/
HOST: 239.255.255.250:1900
CACHE-CONTROL: max-age=900
Content-Length: 0
```

Figure 49: Enter Network

When a device leaves the network or is no longer available, it will multicast a *ssdp:byebye* message for each previous sent *sddp:alive* message that has not yet expired. If however the device would abruptly leave the network, the max-age expiration value will still enable UPnP devices to eventually conclude that the device and its services can no longer be accessed.

```
NOTIFY * HTTP/1.1
NT: upnp:rootdevice
USN: uuid:cbc7208e-b395-4e61-acd0-9883071ef021::upnp:rootdevice
NTS: ssdp:byebye
HOST: 239.255.255.250:1900
Content-Length: 0
```

Figure 50: Leave Network

10.4.2 Discovery

M-SEARCH messages are search requests that are multicasted using HTTPMU and include a *Search Target (ST)* for a particular device, device type, service, service type or all devices. Moreover, *sddp:all* can be used as ST value to discover all available root or embedded devices and services on the network.

Devices or services that match the ST value should wait a random number of seconds, up to the MX indicated value, in order to avoid flooding the requesting device with search responses from multiple devices.

```
M-SEARCH * HTTP/1.1
ST: upnp:rootdevice
MX: 10
MAN: "ssdp:discover"
HOST: 239.255.255.250:1900
```

Figure 51: Search Root Devices

M-SEARCH responses messages are unicasted using HTTPMU and follows the same pattern as for the aforementioned *NOTIFY* advertisements. In case devices wish to interact with the responding device, the UPnP device description referenced by *LOCATION* header will be required.

```
HTTP/1.1 200 OK
LOCATION: http://10.0.2.3:64270/
EXT:
SERVER: PPC2002, UPnP/1.0, Intel MicroStack/1.0.1186
USN: uuid:MJWTWVBTBMRIZCOOION::upnp:rootdevice
CACHE-CONTROL: max-age=1800
ST: upnp:rootdevice
```

Figure 52: Search Response

10.4.3 Subscription

Subscription messages are sent using HTTP TCP/IP and contain the service that is subscribed to, and a *HOST* header that references the location the message is send to. The *CALLBACK* defines the location that event messages must be sent to.

```
SUBSCRIBE /SwitchPower/event HTTP/1.1
NT: upnp:event
TIMEOUT: Second-300
HOST: 10.0.2.3:8085
CALLBACK: <http://10.0.1.1:9696/KYRJDHIGICCOYNCKYWC/SwitchPower.0001>
Content-Length: 0
```

Figure 53: Subscribe

If the subscription request has been processed, a subscription response is sent back to the subscriber using HTTP TCP/IP. The *Subscription Identifier (SID)* header is a unique id.

```
HTTP/1.1 200 OK
SERVER: PPC2002, UPnP/1.0, Intel MicroStack/1.0.1181
SID: uuid:1
TIMEOUT: Second-300
Content-Length: 0
```

Figure 54: Subscribe Response

10.4.4 Event Notification

Event notification messages are sent using *HTTPU* and contain a *HOST* header to which the notification is sent to, and a *NOTIFY* header defining the method that notifies the subscriber about the particular event. Moreover it contains the *Subscribers ID (SID)* and a SEQ header that defines a sequence number in order for subscribers to distinct events.
In addition, the event value is also enclosed and attached as a *SOAP* message.

```
NOTIFY /KYRJDHIGICCOYNCKYWC/SwitchPower.0001 HTTP/1.0
HOST: 10.0.1.1:9696
Content-Type: text/xml
NT: upnp:event
NTS: upnp:propchange
SID: uuid:1
SEQ: 1
Content-Length: 156

<?xml version="1.0" encoding="utf-8"?>
<e:propertyset xmlns:e="urn:schemas-upnp-org:event-1-0">
  <e:property>
    <Status>true</Status>
  </e:property>
</e:propertyset>
```

Figure 55: Event Notification

A subscriber must respond within 30 seconds to the eventing device, after it has correctly received the event notification.

```
HTTP/1.1 200
OK
Content-Length: 0
```

Figure 56: Received Event Notification Confirmation

10.4.5 Service Request

Finally, service requests are issued as a SOAP message that is send via HTTP TCP/IP. The request message contains a POST header which defines the control URL for the requested service and a SOAPACTION header that defines the action to be performed. In adition it contains a HOST header that defines the location to which the message is sent to.

```
POST /SwitchPower/control HTTP/1.1
SOAPACTION: "urn:schemas-upnp-org:service:SwitchPower:1#GetStatus"
CONTENT-TYPE: text/xml ; charset="utf-8"
HOST: 10.0.2.3:8085
Content-Length: 282

<?xml version="1.0" encoding="utf-8"?>
<s:Envelope s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <u:GetStatus xmlns:u="urn:schemas-upnp-org:service:SwitchPower:1" />
  </s:Body>
</s:Envelope>
```

Figure 57: Service Request

Responses to service requests are also enclosed within a SOAP message. The body of the soap message contains the service reference and the status value of the service.

```
HTTP/1.0 200 OK
EXT:
CONTENT-TYPE: text/xml
SERVER: PPC2002, UPnP/1.0, Intel MicroStack/1.0.1181

<?xml version="1.0" encoding="utf-8"?>
<s:Envelope s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <u:GetStatusResponse xmlns:u="urn:schemas-upnp-org:service:SwitchPower:1">
      <ResultStatus>0</ResultStatus>
    </u:GetStatusResponse>
  </s:Body>
</s:Envelope>
```

Figure 58: Service Response

10.5 Certificates

The following commands depict the creation of the TEAHA Root CA Certificate, adding the certificate to the store and the export of the private key to the (PFX) certificate.

```
makecert.exe -n "CN=TEAHA Root CA,O=The European Application Home Alliance" -pe -ss my -sr
LocalMachine -sky signature -m 96 -a sha1 -len 1024 -r TEAHA_Root_CA.cer

certutil.exe -f -addstore Root TEAHA_Root_CA.cer

certutil.exe -privatekey -exportpfx "TEAHA Root CA" TEAHA_Root_CA.pfx
```

Figure 59: TEAHA Root CA Certificate

The next example shows the creation of the TEAHA Device Certificate, and export of the private key to the (PFX) certificate.

```
makecert.exe -n "CN=TEAHA Device,O=The European Application Home Alliance" -pe -ss my -sr
LocalMachine -sky signature -m 96 -in "TEAHA Root CA" -is my -ir LocalMachine -a sha1 -eku
1.3.6.1.5.5.7.3.1,1.3.6.1.5.5.7.3.2 TEAHA_Device.cer

certutil.exe -privatekey -exportpfx "TEAHA Device" TEAHA_Device.pfx
```

Figure 60: TEAHA Device Certificate Signed by TEAHA Root CA

The following commands enables the created TEAHA Root CA and Device certificates to be imported into the Java KeyStore.

```
keytool –importcert –file “TEAHA Root CA.cer” –alias “TEAHA Root CA”
keytool –importcert –file “TEAHA Device.cer” –alias “TEAHA Device”
```

Figure 61: Import into Java KeyStore