Service Oriented Interface Design for Embedded Devices

Rainer Glaschick
Siemens Business Services
Paderborn, Germany
rainer.glaschick@c-lab.de

Brigitte Oesterdieckhoff
Paderborn University
Paderborn, Germany
brigitte@c-lab.de

Abstract

In the SIRENA project, the service oriented approach is used to connect devices using Web Services or Universal Plug and Play (UPnP) In this environment, resources are scarce, and near real time behaviour and high reliability without user intervention is needed. At least in our context an interface design for Service Oriented Architecture should not use a classical object oriented method decomposition and export these methods as service actions, although this approach is suggested by corresponding tool support. This is not only be highly inefficient, but very often ignores the unreliability because of the real concurrency that is always present in networked applications.

The article analyses the possible problems in interface design for service oriented control, explains differences to object oriented programming and refers to well-known techniques to avoid the issues described.

Keywords*: Service Oriented Architecture, SOA, Web Services, service oriented computing, object oriented programming, embedded devices, Universal Plug and Play, UPnP.*

1 Introduction

High integration provides today's microprocessors with an interface for wired and wireless networks at nearly no additional cost, so that communication between devices is more and more based on these universal multifaceted interfaces, using standard IP protocols. Required is thus a universal and simple cooperation scheme between devices in a networked environment. Several solutions have been successfully used, starting with *Remote Procedure Call* (RPC), continuing with the *Common Object Request Broker Architecture* (CORBA), *Distributed Component Object Model* (DCOM) or using *Remote Method Invocation* (RMI) like in JAVA.

Currently, so called *Service Oriented Architectures* (SOA) become the standard for data exchange in a networked environment. In the SIRENA [1] project, these are used for inter device communication and control for embedded devices in resource restricted and nearly real time environments. *Universal Plug and Play* (UPnP) [2] is a zero-configuration protocol for the control of appliances in a home environment, that could be regarded as a specialized early adoption of a SOA. Because of several architectural problems with UPnP described in an earlier article [4], Web Services were used for newly designed interfaces in the SIRENA context.

The creation of a SOA using Web Services is already supported by tools for nearly every broadly used programming languages, that allow the methods of an object or the functions of the programming language to be exported as services. This good tool support suggests that a Service Oriented Architecture for a networked system is nothing more than creating the necessary objects as usual for object oriented programming, and exporting these as service actions.

However, at least in our context, a classical Object Oriented method decomposition and the export of the corresponding methods as service actions can be highly inefficient and give noticeable problems in the context of networked embedded and resource restricted devices. More important is the unreliability that may inadvertently be introduced because of the real concurrency that is always present in networked applications. It is claimed that SOA actions should be stateless, and if this is really the case, no concurrency problems will occur. But really stateless services, like queries to unchanged databases, are rather seldom. In particular, the common technique used in Object Oriented method decomposition, characterizes by a large number of GET and SET methods, is inherently stateful and unsuitable for a Service Oriented Architecture.

As an application example in the Sirena project, a distributed media system for hotels or homes use is selected. Media replication and streaming used on embedded devices in a nearly real time environment, provides a good example of problems with real concurrency in networked applications that have to be solved.

This contribution is organized as follows: Section 2 gives some related work. Section 3 describes existing problems if SOA and Web Services are used in a networked system of embedded devices. In Section 4 object oriented design and service oriented design is considered. Section 5 gives some useful techniques to overcome the described problems, illustrated by some examples from

the SIRENA home demonstrator in section 6. A short overview of the scenario from SIRENA is given in Section 7. Finally, Section 8 provides conclusions and outlook.

2 Related Work

For the subject of Service Oriented Architectures in general, a large number of articles [11], [12], [13] as well as books are available. The W3C has published a rather detailed overview of the *Web Services Architecture* [17]. In *Elements of Services-Oriented Analysis and Design* [7], several subjects on architecture and design are explained. In the subsection *service granularity*, there is referred to *the advice to model coarse grained* as in several other high-level papers on Web Services, but a more detailed discussion could not be found.

In *Web Services Are Not Distributed Objects* [5], the relation is discussed on a high level. This comparison is not only necessary because the acronym SOAP once stood for *Simple Object Access Protocol*. There exist several tools, in particular in interpreted programming languages like PHP, Perl and Python, and of course in JAVA, that allow object methods to be exported as Web Services. A more general treatment can be found in [10], [6] and [14]), and a rather specific advice is found in [15]. The *gSOAP* [3] toolkit allows to generate stubs and skeletons for C and C++ for Web Services, using the *Web Services Description Language* WSDL [18]. The article *Web Services and Business Transactions* [16] and *Reliability of Composed Web Services* [9] are related to the coarser transactional view of business transactions.

The authors are coauthors for an article [4] submitted to the INDIN05 conference, where the emphasis lies on the relation to UPnP [2] and the architecture of the Home Demonstrator in the SIRENA [1] project.

3 Analysis

Using a Service Oriented Architecture (SOA) to connect various devices and applications, some problems may occur that are not present in a local solutions. These are analyzed in this section, and countermeasures are given in the next section. Note that these problems are not present in a large number of web service applications, in particular those common examples like querying a relatively static database like that of a book merchant, and where out-of-date results are not disastrous, in contrast to real time applications.

3.1 Concurrency

Connecting devices by networks means an environment with real concurrency. Already the exchange of a single packet takes several hundreds or thousands of processor cycles, so that incoming requests and replies are not predictable in time. Collisions may occur and increase the time even more.

From the server's view, network request arrive unsolicited, at any time from clients not known in advance, and are thus completely unpredictable. It is clearly not useful to restrict communication to devices known in advance; this destroys the major benefit of the service oriented approach, its flexibility.

The best way to avoid concurrency is to make the methods thread-safe, without using semaphores and locks. This will result in passing state information to the caller explicitly, but cannot be avoided, as far as we can see now.

Stateless Services The general answer to concurrency problems is to use stateless services, which is sometimes claimed as typical for service oriented architectures.

As long as the services are queries on data, and that either each query is self contained or the data is effectively immutable, this model works fine. Another example, equally pro and contra, is the Network File System (NFS). It works fine, unless file locking is necessary, in which case the service is no longer stateless in a strict sense, even if an interface can be provided that looks like stateless. In particular, the NFS service benefits from the fact that the data is essentially unstructured, as seen from the service. This means that there are no interdependencies to be observed; the file serve simply can supply and change parts of the byte stream called file.

However stateless services are the exception. Effects that prohibit the use of stateless services are:

Large Data If the data exceeds the buffer space of the receiving device, it has to be transferred in several calls. While the stream oriented protocol TCP allows a nearly unlimited data transfers size, fragmentation must be processed on application level, which is contradictory to the service oriented approach. And small devices cannot set aside large buffers to store the whole request or reply before passing it to the XML parser or the application. Thus, interfaces must be designed in such a way that the amount of data is predictable.

It is admitted that this type of problem does rarely occur if fine-grained object oriented interfaces are used, as these are bound to transmit only a single atomic piece of data with each call. However, these use patterns tend to be inefficient, so aggregating is a solution, that on the other hand may create very large data volumes to be avoided.

Structured Data Sometimes, the data to be transferred is tree structured. The standard solution is to provide an interface for tree traversal, where the tree is traversed using elementary calls, and often rebuilt in the caller's memory from these calls. This and similar pattern are often used when the data model is such that the exact amount and structure of the data is not known in advance.

While this is seldom the case for low-level devices, it occurs rather often with services that aggregate data from various devices, which do appear and vanish in time.

Then, the database or data tree reflecting the states is changing in time.

It is obvious that the tree or list traversal use pattern depends on an immutable tree during traversal, because otherwise there may be links to nodes that do no longer exist when referred to.

Moreover, many trees have semantic relations between nodes and arcs, so that even if the tree traversal itself is programmed in a safe way, the resultant copy of the tree may be inconsistent with regard to the semantic contents; and this will normally result in failures of the routines using the copies.

Context dependent setting If the device is not only read, but also to be set to a certain state, this setting most often depends on the state of the device. This state has been read prior to the decisions of the controlling application in order to determine the new parameter. Very often, the new setting is only valid as long as the device has not significantly changed its state. In particular, the values read out before constitute a set that has to be the same when the parameter is set. If, however, another controlling application has meanwhile acted, the double action may be disastrous.

An example is a redundant file storage. As long as at least one file is left, the other replicas may be deleted. If two controlling applications both see two files, and elect to delete one, but both not the same, the system ends up with no file. The same can occur if two valves can be operated with similar results, and the controlling applications does not want to open both at the same time. Another example is that the same process parameter is set to two different values by different controlling applications, the one being earlier loses control. Yet another example is a device connected to the real world, that counts e.g. the number of incoming people.

Let us further explain the case by a more detailed example: An information display has a device to identify persons, e.g. via face recognition or RFID transponders. The ID is polled (or sent by eventing) to the server, that selects a message and sends this back to the information device. It must be assured that the message is not displayed if the person has left the device in the mean time. This is not difficult to be assured, but it requires proper interface design carrying enough information together with the message to allow the device to check the transactional consistency of the control message.

So there are two reasons when there must be some kind of transactional support: Either if the device can be controlled by more than one application concurrently, or if the process value may change until the new setting has reached the device.

3.2 Statefulness

As mentioned above, service actions should be stateless. We have two kinds of "state" in this context, following from the discussion of the previous paragraph.

First, we have a closed and implied context between calls, where a first call sets the context for the second call; i.e., the semantics of the second call depend on the parameter values of the first call. This may be called strong statefulness.

An example is a case where the first call sets the operation code, and the following call contains the parameters. Thus, the first call sets a certain state, that must be unchanged for the second call to function correctly. This strong statefulness can easily be avoided by combining both calls into a single call, and should nearly never be used for SOA interfaces.

Second, as already described, the state of a device is first examined, then the action determined and called, but may give undesired behaviour if the state has changed meanwhile. We may call this soft or semantic statefulness, and this type occurs very often and is mostly hidden, i.e. not visible from the parameter specification. It is, however, to a large extent visible if pre- and postconditions are specified.

3.3 Efficiency

Sending a block of data through the networking layers from one machine to the other always involves several hundred to several thousand instructions. If XML encoding is used, this overhead will even be larger. So sending a plain number via the network is several magnitudes slower than an assignment inside a programming language.

This means that it is highly inefficient to transfer a row of numbers each in a separate networking call. This would be the case if for each of these numbers a service action is used, which is simple to specify and to implement. From the efficiency point of view, it is better to have a service action even with a complex parameter structure, because the evaluation of what to do is always much more quicker than sending a second message. For standard single machine programming, it is still nearly always better to do two subroutine calls to different functions than to have one single subroutine that must figure out the cases.

Because in the early days of computing, a call to the operating system was relatively expensive, both in absolute time and and compared to the total CPU consumption of the process, some of the techniques used then can be used again, as will pointed out in Section 5.

3.4 Eventing

Eventing means that a client can register with the server to receive asynchronous messages in case of state changes (in the broadest sense). Effectively, server and client exchange roles then, because the server is sending a SOAP (Simple Object Access Protocol) packet to the client parallel to the normal control flow of the client. Eventing can be implemented even without dedicated support; the server just has to publish an action where the client can register for the event. Normally, middleware is hiding this part of the service.

The major difference is that while normal services need

to detect or configurate the server's address and interface description either statically at design time, via directory services or even dynamically via discovery. The eventing callback does not need all these, because the client can register directly with the server.

Eventing saves the client from polling the state of the server in regular interval, and speeds up the detection of server state changes. It has even been seen as a solution for the consistency problem. In UPnP [2], the state of the server should be completely determined by a row of simple state variables, and UPnP eventing signals any state change. Thus, a client (called control point in UPnP) could use fine grained GET and SET methods, because it will be alerted asynchronously if the state changes.

However, the proper synchronization may be very difficult, and care must be taken not to run into logical loops. This can happen if the client sets a variable, receives the state change event, starts an evaluation, determines new state values, changes these, and in return gets a state change event again, and so on.

4 Object Oriented and Service Oriented Designs

Object oriented design has some principles [8] that do not fit into the service oriented approach. These are:

Inheritance Object oriented design requires object hierarchies with inheritance. Service oriented interfaces and in particular SOAP messages are not at all supporting inheritance; they are closer to remote procedure calls than to any object oriented system. From this observation alone, it is clear that service oriented interfaces are not object oriented programming over networks.

Data Abstraction and Encapsulation One of the earliest software technologies was data abstraction, later also termed data encapsulation or data hiding as the concept of an object arose. It means that access to an object's data is done only via method invocation, so as to hide the internal representation. This concept has been proved to increase the reliability with only minor performance penalty.

As a result, it has become best practice to use objects with a row of methods to GET and to SET values inside the object. The implementer starts with simple GET and SET methods that are essentially compiled to an assignment. If later more checks become necessary, they can easily be implemented. To query or set up and modify an object, a row of GET and SET method calls is then used. These GET and STE methods should be made as small as possible, thus the design is often termed *fine-grained*.

Not astonishingly, the same technique is commonly used for service oriented interfaces; and in particular because there exist tools to export the interface methods of an object as actions for a Web Service. So often service definitions have a large number of actions, starting with either GET or SET. If the modern tools are used that map the methods of an object to Web Services, the programmer normally starts to define the objects as usual, and then does not realize that using the object as a service, will require a series of actions calls, and hence risk a severe performance trap that will nearly never be found out during tests; and if it is, hard to repair.

This is not a great issue if modern high performance server and workstations are used, because few applications require response times in the microsecond range instead of the millisecond range. And most critical servers have to process several requests in parallel anyhow.

For embedded devices in industrial control and other time critical applications, however, additional delays of several milliseconds can severely spoil the performance, because an embedded device has nothing to do in between and absolute time really matters.

As mentioned above, the fine-grained GET/SET design also opens consistency problems.

Binding One can always pass as the first parameter a pointer to the object to be acted upon, and use type checks to assure that each function is called only with the appropriate object reference. Object oriented programming language implementations ease this technique passing the address of the object implicitly and transparent for the programmer to the methods.

This binding of methods to the object can successfully applied to services without significant penalty. The server action is provided as an object, and the methods are invoked by the middleware necessary for the SOAP format anyhow. The client instantiates an object representing the remote web service, and uses the methods as usual, while the middleware cares for the SOAP packaging etc.

However, this model is in general static, in that the service description is used to generate objects and templates at coding time, and has not provision to dynamically adopt to changing interface descriptions over time.

Overloading While not strictly object-oriented, most such languages provide overloading for functions. This concept is also conceptually no problem. On the client side, the parameters are converted to XML phrases anyhow, and include type information that is mirrored from the signature. On the server side, the the type information in the SOAP message can be used to select the method that fits to the parameter types.

In so far overloading might the be a good technique to provide a rich set of actions in a fairly transparent way. In particular, then GET/SET methods could be provided for each single variable as well as for any combination of variables. However, this subject has not yet be evaluated in our project.

5 Useful Techniques

Several well-known techniques can be used for SOA designs to circumvent the above described problems. Some of these techniques are easy to apply because SOAP has the option to use more than one output parameter and also combined input-output parameters, that have a slightly different semantic than the call-by-reference to which they are often mapped, i.e. it is a call by copy-overwrite.

5.1 Concurrency

Locking Locking is useful only in close coupled systems like an operating system, where locks are quick and stale locks can be either automatically cleared by the operating system if the process terminates, or the state of the whole system can be inspected rather easily. Locking would be a solution if TCP connections were used and hold during the lock, as the TCP layer uses low-level messages to assure that both ends are alive. Because SOAP over TCP/IP in its standard implementations does not keep connections between action calls, locking must be above this level and would need rather long timeouts and be a source of denial-of-service problems.

Compare-and-Swap With Compare-and-Swap, the function to set a new value also carries the old value. The new value is set only if the old value is still present. Otherwise, the setting of the new value is not done, and an error reported back. Thus, the controlling application can be sure that the controlled device is in the assumed state.

In particular, if the device is modelled by a small number of simple variables, this technique is extended to all these variables. One possible solution uses a single function to obtain the values of all variables, and other functions to set values, providing the values obtained with the first function as check values. Even a single call can be used, with an additional flag to indicate that the check parameters are valid, and a flag to indicate which set parameters are valid.

An example is the control of a valve. The controlling application has obtained the current mode of the process, and in particular the current position of the valve. The new position is calculated and then a set call used with two parameters: bool SetValve(position old, position new).

If another controller has modified the position in between, the situation is easily detected this way without any complex locking mechanism.

In a less clumsy variant, only the values that not set are check values. Here, the parameter list includes all parameters as Input-Output parameters, and a row of flags to indicate which of these are to be set. The others are checked.

In a variant of this technique, INOUT parameters are used in conjunction with a row of flags, indicating which of these are really INOUT, and which are OUT only.

State Indicator In most cases, the best technique is to have a state indicator. This may be counter that is incremented with each state change, or a timestamp of the last change. Using zero as a special value, the interface could be made up in a highly uniform manner using consistent semantics (see example *GetDistinctMediaList* in Section 6).

Each action has as its first parameter the state indicator, formally an input-output parameter. If it is zero, this means that no check has to be made, but the current value returned. In general, this will be a call to obtain some state variables. If the state parameter is not zero, it is checked to be still unchanged, and the action immediately returns an error if not. Thus, a single service action can be used when the device has only a small number of plain state variables. The second parameter is a row of flags, followed by an input-output parameter for each state variable. The flags indicate which parameter is effectively an input parameter, i.e. which parameter is to be set:

- If none of the flags is set and the state indicator is zero, the state is queried.
- If none of the flags is set and the state indicator not zero, the device can be checked to be in an unchanged state.
- If a flag is set and the state indicator is zero, the device is set unconditionally. This case may be excluded in order to tease bad programmers that are not using the state variable at all[1]
- If a flag is set, and the state indicator not zero, a normal change occurs if the state indicator is correct.

In the UPnP [2] media server, the tree traversal used to obtain the media list has a state indicator to ensure the consistency of the media tree is kept. However, it is an output parameter, and it is the duty of the caller to check it with each call. This choice should be avoided, as the check code is duplicated for each call, and it its likely to be forgotten in the first coding round and never installed afterwards.

The state indicator could help avoiding loops triggered by eventing, but this has not yet been evaluated in our project.

Transaction Number This technique is similar to the state indicator, but controlled by the caller and used when a sequence of calls cannot be avoided. In this case, the caller generates a different (random) transaction number for each series and includes it in each call. For clarity, action brackets like *open* and *close* should be used, but the transaction number still included in each call.

There are two possible semantics:

[1] but it does not help, because this kind ignores all error codes and simply gets the state variable before each other call and ignore the possibly new state values

1. The transaction number acts like a lock, and other open calls are refused while a transaction is open. Not recommended, because a timeout is needed, and the recovery semantics become complicated.

2. All changes between *open* and *close* are buffered, and applied as a single operation when the *close* call is received.

In the second case, and depending on the memory available, a full merge may occur, or any previous transaction immediately abandoned when a new one is opened.

6 Examples

In this section, some Examples from the SIRENA [1] home demonstrator, shortly described in the next section, are explained in order to illustrate the above techniques.

6.1 Get Distinct Media List

A distributed peer-to-peer database collects all media entries in a flat list. This list can have arbitrary length and should always be consistent. Instead of a linear list, a tree could have been used.

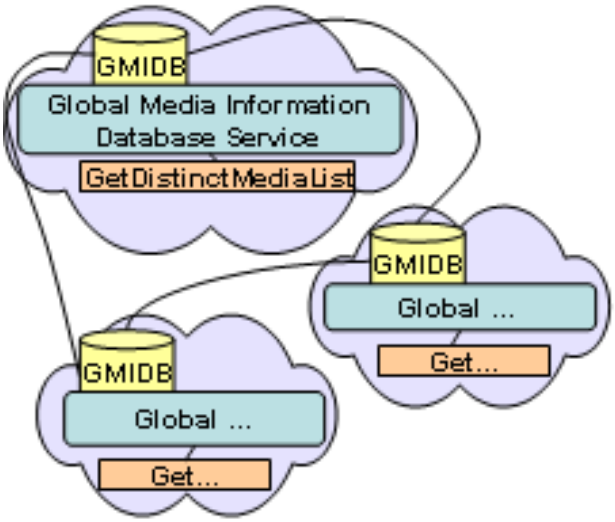

Figure 1. Action: GetDistinctMediaList.

The action call retrieves a list of all media, where replica are shown only once. The following table shows the parameters of the action:

Name	Direction	Type
GMIDBStateID	INOUT	NonNegInt
FirstEntryIndex	INOUT	NonNegInt
ReqNumberOfEntries	INOUT	NonNegInt
TotalNumberOfEntries	OUT	NonNegInt
DistinctMediaList	OUT	DistMediaList

Operation: Normally, the GetDistinctMediaList has both first parameters, *GMIDBStateID* and *FirstEntryIndex*, set to zero to indicate a new transaction. The third parameter *ReqNumberOfEntries* indicates the maximum number of entries to be returned in order to avoid buffer problems.

On return, a *GMIDBStateID*, the index *FirstEntryIndex* of the first entry in the *MediaList*, and the number *ReqNumberOfEntries* of entries in the *DistinctMediaList*, are returned.

The return value *TotalNumberOfEntries* allows to check if continuation calls are necessary, i.e. the number of entries in the database exceeds the *ReqNumberOfEntries* given in the call. In this case, the continuation call should provide the *GMIDBStateID* as returned, *FirstEntryIndex* as returned *ReqNumberOfEntries* +1, and an appropriate *ReqNumberOfEntries*. Requests may, however, be overlapping or not contiguous as desired by the application.

The *GMIDBStateID* ensures that all requests with the same *GMIDBStateID* are coherent and return the same data for the same index. If, the database was changed meanwhile, an error return will occur and no data returned.

The ReqNumberOfEntries may be zero on input to inquire the number of entries in the database without returning any entry. On output, it may be less that the input number even if more entries are available, if e.g. the database had not enough buffers.

6.2 StreamControl

Other service calls allow to open a stream between a server and a renderer. Such an open stream can be examined and controlled by a single action:

Name	Direction	Type
StreamID	IN	Integer
DoPosition	IN	Boolean
DoPlaySpeed	IN	Boolean
DoStartStop	IN	Boolean
StartStop	INOUT	Boolean
Position	INOUT	Integer
PlaySpeed	INOUT	real
EndPosition	OUT	Integer

Operation: If *DoPosition*, *DoStartStop* or *DoPlaySpeed* are set, the correspondingly named parameters are used to set the position, the playing speed and to start or stop. The service always returns the current values of *Position*, *StartStop* and *PlaySpeed* after the operation has been done. Thus, leaving all three *DoXX* parameters to false, the current state is returned. The *EndPosition* allows a relative indication of the the position in a user interface.

6.3 ReplicaUnregister

These two cooperating actions looks very simple yet use two variants of the state id technique.

The transaction problem behind is that two replication agents could at the same time decide to delete a replica, and if these two were the last ones, the user may have none after all.

In the peer-to-peer database, each instance has a list of local (homed) media, and caches a list of the media in other instances. In order to avoid this happen, there is not only the StateID of the database, but also the assumed number of replica indicated. Then, the called action can assure that after the operation, the remaining number of replica is what was intended. Note that for the P2P database, there are two actions, the first called by the replication agent to remove the local copy and the other one used as intra-database call to remove the cached entries.

ReplicaUnregisterHomed:

Name	Direction
GMIDBStateID	IN
MediaID	IN
ReplicaID	IN
MediaControllerID	IN
CopiesRemaining	IN

Operation: If the *GMIDBStateID* is correct, and the other (redundant) parameters match the database, the number of replica copies is recalculated and must be one more than the last parameter. Then, all other GMIDB instances are requested to remove the mirroring entries. If all report success, the homed entry is removed and the action returns without error. Then, the calling media controller may delete the replica finally.

ReplicaUnregisterMirrored:

Name	Direction
MediaID	IN
ReplicaID	IN
MediaControllerID	IN
CopiesRemaining	IN

Operation: After a check that the entry exists in the mirrored part, it is removed. Then, the number of copies with the given *MediaID* is locally determined using the homed and new mirrored part. If the result does not match *CopiesRemaining*, an error is returned, telling the calling database instance to keep its copy which is the homed copy.

The pair (*ReplicaID*, *CopiesRemaining*) can be regarded also as a transaction id, generated by the caller from previous inquiry calls. However, no open/close pair is necessary here.

7 Scenario

The examples in the previous section were drawn from the Home Demonstrator out of the SIRENA [1] project. A more detailed description can be found in [4].

Purpose of the demonstrator, illustrated in Fig. 2, is to show that the Web Services based SIRENA middleware is well suited for applications with near real time requirements (like media streaming) and embedded devices.

Several subnetworks are connected on a peer-to-peer basis. In each subnetwork we maintain the following components:

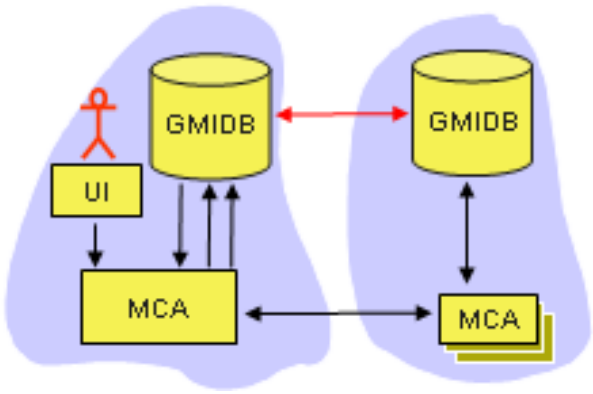

Figure 2. Part of Home Demonstrator

- **Media Control Services** (MCA): Provided as a central control instance, running as a device. A Media Control instance may be considered as a WS-enabled management instance for devices. The MCA communicate with each other using Web Services on a peer-to-peer basis.
- **Global Media Information Data Base** (GMIDB): A peer-to-peer connected service managing content references for the whole network. Each subnetwork maintains one GMIDB instance which provides this information service via Web Services to one ore more MCA in its subnetwork.
- **User Interface** (UI): Also connected via Web Services, the user interface is decoupled from the MCA in order to be able to run on any device, e.g. a handheld, mobile phone etc.

Not shown in the picture are additional services for resource allocation, media selection. Also not shown is a media replication service, that will replicate media in such a way that there is a high probability that a user who wants to see a media will have a path to the server containing that media that has the necessary resources. Thus, there is a lot of concurrency in this network, where the media replication service replicates media or tries to remove replica, while the user will select media from different subnetworks. Playing the media is performed by (not shown) existing UPnP [2] devices, even if in future versions these devices could be controlled via Web Service Interfaces too.

8 Conclusion and Outlook

A number of potential problems in using web services, in particular for embedded devices and real time services, were analysed, set in relation to object oriented programming, and some techniques to avoid the problems are given. From our point of view, the following items should be checked for an interface design using Web Services with SOAP or similar situations.

- If methods of objects are exported as services, these methods must be thread-safe without locks and semaphores.
- Avoid the fine-grained getter/setter technique common in object oriented programming
- Combine similar functions into a common service action, using parameters to control the variants
- Use state indicators to ensure coherent states and information
- Check state indicators in the server, not in the client
- Check for soft, semantic state dependency of actions
- Use parameters to obtain slices if the final data may overflow buffers in the caller or server
- Beware of loops created by eventing, in particular if the event triggers a state evaluation

Acknowledgement

Parts of the work described herein was funded by the German Federal Ministry of Education and Research (BMBF) within the ITEA-SIRENA project [1].

References

[1] SIRENA, *Service Infrastructure for Real-time Embedded Networked Applications*, Eureka Initiative ITEA (Information Technology for European Advancement) project, (01ISC09E), http://www.sirena-itea.org

[2] UPnP Forum. http://www.upnp.org/

[3] van Engelen, gSOAP & Web Services R. A. Van Engelen, K. A. Gallivan: *The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks.* CCGRID '02: Proc. 2nd IEEE/ACM Int. Symp. Cluster Computing and the Grid, IEEE 2002

[4] B. Oesterdiekhoff, C. Loeser, I. Janich, R. Glaschick; *Integrative approach of Web Services and Universal Plug and Play within an AV scenario.* Submitted for INDIN05, Perth, Australia.

[5] W. Vogels; *Web Services Are Not Distributed Objects.* IEEE Internet Computing, Nov/Dec 2003, pp. 59-66

[6] M. Gaedke, H.W. Gellersen, A. Schmidt, U. Stegemller, W. Kurr; *Object-Oriented Web Engineering for Large-scale Web Service Management.* Thirty-Second Annual Hawaii International Conference On System Sciences (HICSS-32) on the Island of Maui, USA, January 5 - 8, 1999. *http://www.teco.edu/ gaedke/paper/1999-hicss32.pdf

[7] O. Zimmermann, P. Krogdahl, and C. Gee; *Elements of Service-Oriented Analysis and Design. An interdisciplinary modeling approach for SOA projects.* http://www-106.ibm.com/developerworks/webservices/library/ws-soad1/

[8] G. Booch, *Object-Oriented Analysis and Design with Applications.* Addison-Wesley Professional; 2nd Edition (September 30, 1993), ISBN: 0805353402

[9] Th. Mikalsen, I. Rouvellou, St. Tai, *Reliability of Composed Web Services.* OOPSLA 2001 Workshop on Object-Oriented Web Services, October 2001 http://www.research.ibm.com/AEM/pubs/web_services_oopsla2001.pdf

[10] N. Milanovic, M. Malek: *Current Solutions for Web Service Composition.* IEEE Internet Computing, Nov/Dec. 2004, pp. 51 ff

[11] Tim Barners-Lee: *Web Services.* http://www.w3.org/DesignIssues/WebServices.html

[12] V. Vasudevan: *A Web Services Primer* http://webservices.xml.com/lpt/a/ws/2001/04/04/webservices/index.html

[13] Hewlett Packard: *Web Services Concepts - a technical overview.* *http://www.hpmiddleware.com/downloads/pdf/web_services_tech_overview.pdf

[14] J. Yang, M. P. Papazoglou: *Web Component: A Substrate for Web Service Reuse and Composition.* Proc. Advanced Information Systems Engineering CAiSE 2002, Toronto, Canada

[15] Sun Microsystems: *Using Web Services Effectively.* http://java.sun.com/blueprints/webservices/using/webservbp.html

[16] M.P. Papazoglou: *Web Services and Business Transactions.* Internet and Web Information Systems, 6, 49-91, 2003 Kluwer Academic Publishers.

[17] World Wide Web Consortium: *Web Services Architecture.* W3C Working Group Note 11 February 2004. *http://www.w3.org/TR/ws-arch/

[18] World Wide Web Consortium: *Web Services Description Language (WSDL) 1.1* W3C Note 15 March 2001. *http://www.w3.org/TR/wsdl/