Collaboration patterns for networked embedded servers

Jean-Michel Douin, Jean-Marie Gilliot

Collaboration patterns for networked embedded servers

J-M. Douin
CEDRIC-CNAM
292 rue St Martin
75141 Paris cedex 03 France
email: douin@cnam.fr

J-M. Gilliot
GET - ENST Bretagne
Technopole Brest-Iroise - CS 83818
29238 BREST cedex 3 France
email: jm.gilliot@enst-bretagne.fr

***Abstract*— In this article, we propose the use of collaboration design patterns to organize communications between numerous embedded devices and cooperative services. The use of collaboration patterns provide a good framework for both synchronous and asynchronous communications with loose coupling. It also allows to organize dynamically device in coherent sets. We also describe a lightweight implementation on embedded web servers with HTTP as protocol, based on a simple url interface.**

I. Introduction

Embedded devices, such as controllers or smart sensors [1] are disseminated in the environment. Because of the need of remote consultation and management, many of them are connected with a network. This is especially estimated for telecontrol applications [2].

As web browser have become a standard interface to communicate and manage embedded devices, and therefore provide a small HTTP Server [3], the web technology will be available on most new embedded devices. This means that the minimal technologies will be TCP/IP, HTTP protocol server and tools that including web servers will possibly be developped in Java.

The question that arise is either to use these minimal communication technologies to develop coherent cooperation between those devices, or develop specific middleware. The problem is then to be able to identify coherent protocol, to enhance the collaboration between servers. We propose the use of collaboration design patterns [4], well known and widely used in Object Oriented Design as a basis for cooperation between many micro-webservers. The idea is to provide schemes of collaboration at design level and to provide a direct and simple correspondence at implementation level.

In order to develop such systems, the communication protocol should be readable by human operator for easy intervention, and should be implemented with a minimal additional layer based on simplest web standards for potential widest use. Special attention must be given to keep extension as small as possible on the server side which is in fact embedded in small devices.

Another communication reason is the cooperation between those devices which enable distributed applications. Those cooperations use mostly specific networks or fieldbuses, for efficiency, cost and habit reasons. On one side, it ensures a good safety level, but on another side, it limits the cooperation to short range neighbours. One may look to the development of distributed cooperation with no limits on distance.

In order to prove the feasability, we develop some of those patterns on some embedded servers. For this, we use a modular web server [5], written in Java, whose initial footprint is very small.

The result of this approach provides a variety of dynamic collaborations schemes between distributed embedded devices, directly available to programmers and users as it respects the standard Java APIs for those local collaborations. Moreover, this is achieved with a minimal influence on current architectures.

This paper is organized as follows. First we define the elements of an architecture based on numerous embedded devices, the nature of exchanges and underlying organization. Then, we show that design patterns may organize the collaboration between devices, and may be viewed as communication components. In the next section, we describe the implementation of the protocol between devices. The last section is devoted to examples of implementation showing the feasability and the flexibility of the communication on one side and the minimal impact on code due to distribution on another side.

II. Managing numerous devices

Usual applications are based on a few servers delivering datas or computation to many clients. When managing sensor or embedded device networks, we face many data providers for a possibly limited set of services/applications. Moreover, different schemes of communications must be handled, Client/Server is one of them, but it is useful to provide event-based communications, also known as pushed models [6]. Additionally, flexibility, such as adding new services that handle a set of sensor datas at run-time is possible.

In this part, we will define what kind of exchange are to be provided for networked embedded devices and then underlying communication styles.

A. Most devices produce datas

Embedded devices tasks are dedicated to their embedding environment. This means they handle a state, corresponding to the state of their environment. This state can be directly given

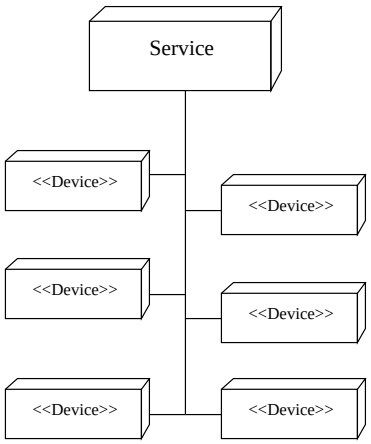


Fig. 1. Embedded devices and services

by sensors, or can be derived from other information (such as control, filtering or data fusion). Services are applications using datas collected for any purpose. This definition is recursive as a service itself may be seen as a (virtual) device. Another definition of device/ service couple is producer /consumer.

Embedded devices are servers able to deliver collected datas at any time, to any service. This means that devices must be known as servers on the network.

We can consider that interactions between device/ service can be handled in one of the following ways:

Gathering datas A service may need to collect datas from a specific set of devices. This may be done periodically or not, but under the responsability of the service. In this case services call each device, that can be seen as clients. Different devices and different services may exit at the same time and evolve (add/remove elements of sets, devices, services).

Asynchronous datas Services may need to be aware of changes of a state of a device. An update of the state of the device, must be send to the service. Services may be added, at run-time, collecting datas. This can be extended to be aware of a set of devices. Services must be ready to accept a call at any time, from any device. This means they must be implemented as servers.

Managing alarms This case is a variant of the preceeding one.

Asynchronously called devices or services must be ready to receive information, i.e. to be called at any time. In our approach, the devices will be declared as servers.

B. Communication styles

As just noticed, two different styles are needed : remote call initiated by the service, or asynchronous calls initiated by the devices. Those basic models of exchange are also defined as pull or push models. Moreover, a lot of entities interested in certain information may vary throughout the time. Collaboration patterns allowing flexibility are therefore a natural basis to manage the communications along the system. Depending on the needs of the services, push or pull models may be needed in the same application. The use of collaboration patterns enables a flexible design, allowing to translate application needs to available technology.

III. Design patterns as collaboration scheme

In order to design the collaboration, we propose to use directly collaboration patterns as available schemes. Those patterns, when properly used have some key advantages.

They provide loose coupling and dynamic update. Subjects and observers don't need to know each other directly. Further objects can be added or discarded during the life time of the application. Some objects may be temporaly down without being noticed directly by other devices or services. It is possible to construct arbitrarly set of device for specific purpose, according to their functionnalities then possibly share some interesting informations.

The use of patterns can be made hierarchically. An observer of a whole of devices may be itself observable in the same application. Observer pattern enforces to be able to make future remote calls to any of the collaboration participants.

To this point, we can conclude that collaboration patterns are good candidates to define globally the collaborations between the devices and the services of an embedded application. The key implementation is now focused on the remote calls to any directions. We will here only concentrate on four classical patterns involving the push and pull communication schemes, namely Iterator, Composite/Visitor, Observer and Model-View-Controller.

The Publish/Suscribe pattern has been proposed as a key pattern to organize the interactions in [7], however we propose to extend this vision to a whole library of patterns.

We don't develop the basic patterns schemes here. The interested reader can consult on this subject many sources of reference. We simply give the page where a pattern is referenced in [4] also known as the Gang Of Four (GOF) book.

A. Collection and Iterator

The first pattern we will consider is the Iterator Pattern (GOF 257) on a collection. This provides a way to iterate and group a set of devices and services. Depending on the dynamic interest, new collections can be constructed, elements may be added or removed, which enabling adaptation. A service needs only to know the collection, asking for an iterator and calling all the objects collected. Those objects don't any additional layer, and the caller don't know them *a priori*. If needed, an object can be referenced in different collections. In this scheme, the caller iterates and makes synchronous calls, like RPC in a distributed system, to get needed processing and/or informations. Gathering datas and polling is then straightforward with an iterator pattern, and well suited. Notice that no specific function is needed on device side except the datas formatting.

B. Composite and Visitor

The Iterator pattern provides a uniform way to acceed through a non structural collection of objects. The Visitor pattern (GOF 331) allows a hierarchical view of the embedded devices. It extends the iterator by taking advantage of specialized objects and by adapting operations to those objects. Hence, it is possible to visit composite objects such as previous collections or other structures and final elements in an uniform way. This also means a more systematic way to aggregate different devices in an application. The specialization can be carried out on three main directions, by taking into account either the nature of the service (sensor reading, command, ...) or the specificity of the device or the selected architecture.

C. Observer

Also known as publish/suscribe pattern (GOF 293), this scheme of communication avoids to poll the state of the subject object, as the observers will be notified of the subject change. As our goal is to manage a large number of devices, in many cases the focus will be on a whole of devices/ information. Hence the subject of the observer pattern will be a distributed collection as discussed in the previous subsection. Neither observers nor objects of the collection need to know directly each other, which enables a dynamic adaptation of both sets. Here in this scheme of computation, devices must handle a notify call to initialies the observers calls. This means that our collection elements has been extended as some "observable like" elements of the Observable Manager in charge of relaying the notification to all the observer's suscribers. Such implementation looks like a two level Observer pattern.

To minimize the exchanges on the network, we can add to the update call the source and the state of the change in the parameters. Notice this is compliant with the definition of the Observer pattern in Java libraries, as value parameter.

D. Model/View/Controller

This pattern (GOF 4) is a generalization of the precedent one and is transparent at the implementation level as the controller just needs to know the model to act on.

E. Underlying Model of distributed collaboration pattern

Our basic patterns can be seen as interaction medium components [8], [9]. In fact, it isolates the communication mechanism in a specific component, describing the exchanges between the classes involved in the collaboration. This component is then itself distributed. In the section concerning the implementation, the component consist in the dotted section on different figures. The aim of communication components is to define interactions between different source of datas and users of those, independently of the different locations. This is exactly what collaboration patterns provide.

F. Application example : temperature and pressure in a building

Let's suppose that every office in a building provide a temperature sensor device hosting a server. This could be easily done by installing a sensor to each computer in a research lab (for example a DS1920 from Dallas/iButton [10]), but be also regarded as connecting to building automation. Different services can then be configured.

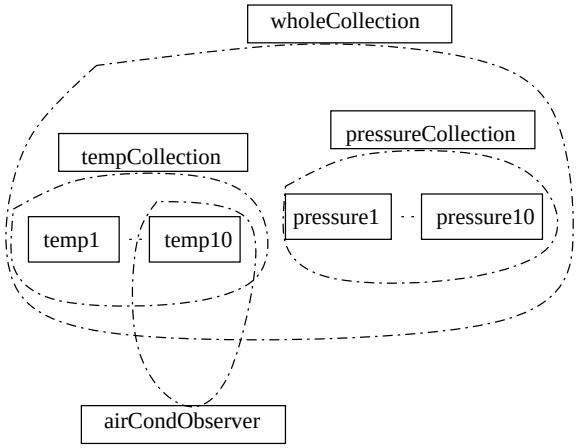


Fig. 2. Example of organisation of devices

- Temperature and pressure knowing and logging of the whole building by polling the whole collection of sensors.
- Supervision of specific areas (for example air-conditioned areas) by creating a sub-collection observer.
- check up of valid and deconnected sensors.

The organisation of the devices used for managing all of them could be as depicted in Fig. 2 where:

- tempCollection would reflect the set of temperature sensors,
- pressureCollection would reflect the set of pressure sensors,
- wholeCollection groups all the sensors as the union of the precedent sets for global gathering or checking,
- airCondObserver groups the areas under specific attention.

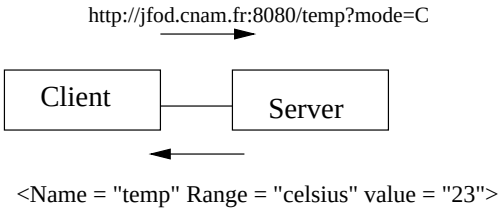
WholeCollection can be visited enabling to make disjointed logging in a single call or to check all sensors in a coherent way. Notice that all sensors may be transparently distributed on common or separated devices.

IV. Protocol Implementation

In order to implement collaboration patterns, we are looking for a minimal additional functionnalities on embedded device side because of their limited ressources. Still any object, being part of a collection, or observer, i.e. any receiver of a message, has to provide a reference to itself. We present now a smallest way to provide a reference and to exchange message in a web context.

A. A device is a HTTP web server

A common trend is to provide on any "smart" device a small HTTP web server, which enables easy reading of the state of the device. This can be achieved with very low footprint [3], i.e. a few kilobytes. This is sufficient to implement directly



the iterator pattern, where the remote call to the object could be simply a HTTP request.

If multiple objects are hosted by the same server on a device, they can be referenced as subdirectories forming their identifiers.

We can say that a message to an object is a HTTP request or even a remote procedure call (RPC) to an object. When parameters are parts of the message, they can be included in the HTTP request as simple parameters.

B. A minimal protocol

1) URL based protocol : a lightweight communication protocol: The web servers will be the objects of the collaboration patterns and will be referenced according to their names or their IP address.

A HTTP request can be handled in a same way as a remote function call, or message passing, where the name will be given as directory extension, in the parameters included in the URL, and the result will be delivered as a simple HTML page. An exemple is given in Fig. 3. It should be noticed that such a call contains implicit parameters which are part of the connexion properties, including the IP address of the caller.

Hence one provides a modular protocol, extensible which doesn't require any new functionality on the server, and is universally supported [11]. SOAP [12] is an based on this basic idea and is a good candidate as a well formalized protocol, but this has not been seen as mandatory at this stage of our work. Moreover, we also tried to avoid any additional software layer, coming from such broader protocol.

2) Adapter as translator to the URL protocol: Using URL based protocol to implement message exchange between servers is an appealing solution. To be usable, we need to define a systematic translation to implement the pattern in a distributed implementation.

This is achieved by a simple translation of all elements of the call in an URL.

The first step is to specify which relations are distributed. The choice of the distribution is out of our scope. Let's just say that an object is located on a device hosting a HTTP webserver. This means it has an IP adress.

The second step is to adapt the messages. Every message exchanged according to the relation, has to be refined using the following rule :

- the goal of the message is the server address hosting the object.
- the name of the object and the method compose the url name
- parameters are translated as couple of ¡name= value¿
- return value is translated as an HTML/XML page. IML, for Instrument Markup Language [13], [14] is a perfectly suitable format for this.

This provides a flexible, low footprint solution, based on a basic service provider. This compares advantageously to middleware like Jini [15].

As a side effect we can observe that such calls can even be made with any web browser.

V. Examples of implementation

As we have seen, collaboration patterns are a powerful way to manage devices as groups or sets. We have shown that a definition allows the location and the call of any object located on an embedded device. The last step is to consider the implementation side on a device. For this, we have developped following examples with a small but extensible server, Brazil. Those pieces of code are just feasability examples.

We propose a java implementation, for its clarity and because embedded java is more and more use with now real time extensions. We also want to take advantage of the fact that some pattern interfaces are proposed for single computer implementations. This last point demonstrates the minimal impact on the distributed code.

A. Brazil : a modular embedded server

Sunlabs proposes a modular, extensible HTTP server, called Brazil [5], [16]. This architecture can run as well on large computers, personnal computers or on embedded computers such as TINIs [17], [18]. It is easy to develop new applications and can be used freely.

The main features of Brazil are :

- *Handlers* which are modular objects, that implementing specific abilities like file services, security services or temperature readings .... Brazil provides a large set of reusable handlers. Using this feature, we developped additional handlers to provide our collaboration patterns.
- *Properties* enables to define and use attribute/value pairs. This make information easily available.
- Support of script languages like TCL or Python, for efficient parsing.
- Security is fully integrated allowing to access control of specific URLs with passwords. We can also restrict access to a server to specific client IP adress, and use the Security Socket Layer (SSL) when needed.
- a server may easily be a portal of different servers, thanks to a redirection handler. This allows combination of servers through a common URL.

It should be noticed that only the used features are loaded. Therefore, servers may be deployed to the needs of the application versus the embedded processors power. This helps to manage the memory footprint of the server, and fit to device or appliances systems capacities.

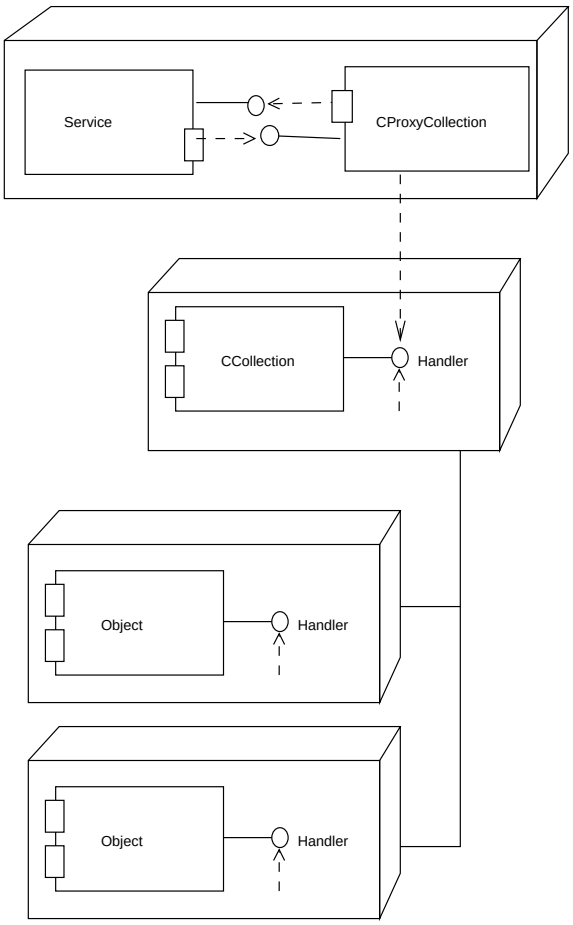


Fig. 4. Possible deployment diagram of iterator/collection pattern

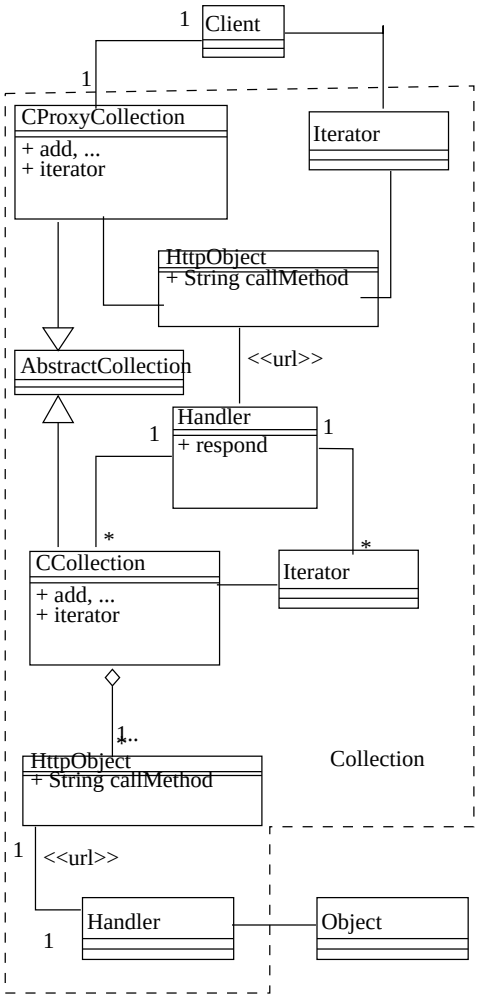


Fig. 5. Implementation of iterator/collection pattern

A callback to an object can be handled by some feature equivalent to a servlet named a handler in Brazil. This enables to define protocols based on URLs, which allows loose coupling in embedded technologies. For our scope, a server just needs to be able to parse an URL request.

B. Collection and iterators

As a first example, we will consider the following structure:

- A service access to a collection through a local adapter called `CProxyCollection`.
- The collection is handled by an instance of CCollection on another processor
- Objects referenced by the collection are freely distributed on brazil servers.

This is resumed on the deployment diagram Fig. 4.

1) Collection: To accede the collection, a service needs a reference to it in order to create a local adapter. This will enable direct access to any method of collected objects, in accordance with the standard method calls. Our adapter implements the `Collection` interface and extends the `AbstractCollection` API standard class, where the only notable difference with a local implementation is the specification of the web address of the actual URLs collection. A call to this adapter will simply translate the call in a HTTP request like the following : `http://server/collection/call/?params`

Notice that the collection may be reached directly with any browser. For example, one can ask the full list of objects referenced with a request such as `http://jfod.cnam.fr:8080/tempCollection/toString/` which will reply with a HTML page containing the full list. If the call is done through the adapter, it will simply remove the HTML tags, in order to deliver the answer.

On the collection side, the HTTP requests are managed by a brazil handler in charge of the collection name. The requests are parsed and corresponding calls with actual parameters are delivered to the collection.

On each site, a remote object is locally represented by a `HttpObject` for HTTP Collaboration Object, containing the object URL and a method to remotelly call a method of the object. Notice that the Collection may itself represented as a `HttpObject` on the client side.

Fig. 5 resumes the final implementation of the iterator/collaboration pattern.

2) Iterator located on Collection site: Asking an iterator of such collection will create a reference for access through the adapter. As a first implementation, we can create the iterator instance on the collection site. Any further access to the iterator will consist in calling an adapter, translating it to a HTTP request, and processing of the response, finally allowing to accede all the objects in the collection. The lifetime of the iterator can be easily ensured with the cookies mechanism, or a unique delivered index value. Should the server fail in a safe way, the persistance of the collection could be also easily guaranteed. This implementation is correct but any access to an object is done thanks three request which is clearly excessive. The only advantage is the minimal resources used on the service side.

3) Iterator located on service site: Another implementation is to make a copy of the list of the objects URLs in the HTML page of the response, allowing to create the iterator near the service. If the collection is constant in regard to the time to traverse the collection, this is clearly a better solution, as long as the memory on the service side is sufficient. In that case the iterator and its state are located on the service site. If one wishes to maintain the iterator coherent with the collection, it could be implemented as observer of the collection.

C. Composite/Visitor

As we have seen the Composite/Visitor pattern is a powerful way to specialize the use according to specific classes. The composite Pattern enables the description of a hierarchical architecture of embedded systems, essential for a complex system. This pattern proposes architectures with a natural view as a list, a group, etc. of embedded systems. In addition, in our context, the composite pattern describes a grammar of embedded systems installed in an environment. This description is proved to be essential with most of systems. As remote objects are represented as HttpObject, this class has to be specialized to provide a hierarchy of elements to visit. This hierarchy provides on one side specialized elements which are specific web servers in our example implementation, and on another side, a hierarchy of collections to provide different ways to group objects. The Composite pattern describes a grammar of the systems used, our example in Fig. 6 defines this grammar in EBNF:

```
HttpObject ::= CCollection | Device

CCollection ::= HttpList | HttpRing

HttpList ::= {HttpObject}

HttpRing ::= {HttpObject}

Device ::= IButton | Tini | JavaCard
```

an expression of the language defined by this grammar is a particular configuration of embedded systems

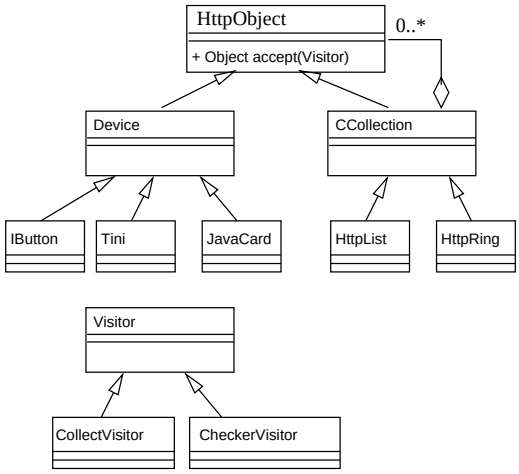


Fig. 6. Example of Visitor

In Fig. 6, we show this example of hierarchy, and two different visitors, one being able to collect the list of the elements in a console, the other checking up status datas. Services can then easily be extended by adding new Visitor classes. For example, we have implemented a new checking visitor for inaccessible devices.

D. Observer pattern

In the Observer pattern, the subject or Observable must be called to add, remove or notify Observers. This means that a HttpObservableManager extending the Observable class and the different methods has to be created on the site managing the observers of the Observable instances.

As a ConcreteObserver (implementing the Observer interface) will be notified by a remote Observable Object, it has to be part of a web server. In the Brazil server, this corresponds to a handler able to receive the URL call and to call update method of the Observer.

As we noticed in section III-C, in order to make numerous device observable, a good implementation is to define the subject as a distributed collection, each element of this collection will simply have to notify the change to broadcast it to all Observers. Fig. 7 resumes the final implementation of the observer pattern. The flexibility, such as adding new services at run-time is possible with our implementation of theses collaboration patterns.

A View in a browser As our implementation is web compliant, it is natural to define browsers as observers. Our current solution is to load an applet from the distant site hosting the Observable. This applet includes a small web server, possibly a Brazil web server or other.

E. Configuration/Initialization of the system

As the different servers involved in the collaborations are passed as parameters in our message, the configuration can be achieved by any external agent. This means that any external broker can initalize the system, or even an operator through any browser.

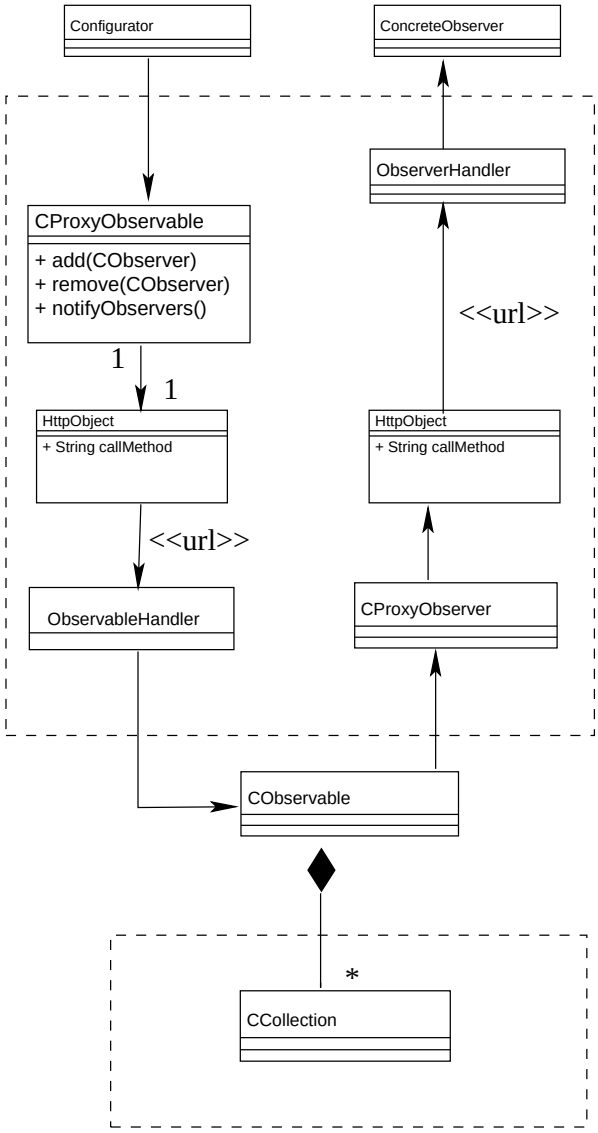


Fig. 7. Implementation of observer pattern

A configuration by default can be achieved with the help of static datas in a configuration file, viewed by the server as initial properties. The Brazil redirection mechanism allows to use its security mechanisms and decouple the deployment and URL signatures. This also ensures a loose coupling with harwdare.

F. Implementation of the application example : temperature and pressure in a building

Let's see now how to implement the example we have described in section III-F. The different collections and the observable collection will be attainable through handlers of Brazil servers.

Let's suppose that temperature sensors deliver their datas on URL `http://www.temp"i".fr/getTemp/` where "i" is an indice variable, and something similar for pressure sensors. A code to initialize the system would look like :

```
CCollection localTempCollection =
 new CCollection(
"http://www.dispatch.fr/tempCollection"));
for (int i = 0 ;
     i < NB_TEMP_SENSORS ; i++) {
 localTempCollection.add(
 new Temp(
"http://www.temp"+i+".fr/getTemp"));
}

CCollection localWholeCollection =
 new CCollection(new
 Gatherer(
"http://www.dispatch.fr/wholeColl");
localWholeCollection.add(
 new Gatherer(
"http://www.dispatch.fr/tempColl"));
localWholeCollection.add(
 new Gatherer(
"http://www.dispatch.fr/pressureColl"));
```

A code trying to collect temperatures on a different site would be almost like a code acceding to local sensors :

```
CCollection localTempCollection =
 new CCollection(new
 HttpObject(
"http://www.dispatch.fr/tempColl"));

for (Iterator
     it = localTempCollection.iterator();
     it.hasNext();)
   float t = Float.parseFloat(it.next());
```

If additional subclasses of HttpObject is defined as proposed in Fig. 8 and if a `CollectVisitor` class implements the "accept" method, access to all the sensors could simply done following way :

```
String Result =
       localWholeCollection.accept(
               new CollectVisitor());
```

This provide a good insight of the potential use.

VI. Conclusion

In this work, we proposed to model interactions between distributed objects on embedded device as collaboration patterns. We have shown that it provides a good framework to organize pull and push models with loose coupling between participants, at logical and hardware levels. The substitution of embedded systems is easy with this approach. The HTTP protocol, broadly used is a *de facto* standard for automation. No additional layer is needed, neither on the device side, nor on the service side where a browser is sufficient in most cases. Flexibility, such as adding new services at run-time is possible with such a scheme.

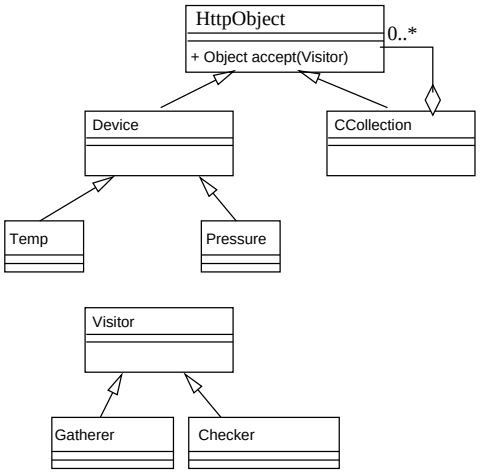


Fig. 8. Implementation example for Composite/Visitor pattern

In section II, we have presented useful collaboration needed for networked embedded devices, then in section III we have seen that standard collaboration patterns fullfill those needs. In section IV, we considered a minimal protocol, when we assumed that HTTP servers are provided on devices. The section V gave details on implementation, showing that our implementation is very close to non distributed API proposed in Java. As examples, we detailed four of the most used, namely Iterator, Composite, Visitor, Observer and its variant Model-View-Controller. The same process has been applied to other patterns such as Chain of Responsibility or Proxy.

The result of this approach provides a variety of dynamic collaborations schemes between distributed embedded devices, directly available to programmers and users as it respects the standard Java APIs for those local collaborations. Moreover, this is achieved with a minimal influence on current architectures.

We believe this may be of great help to manage a system composed of numerous and various embedded devices such as smart sensors.

In the last part we detailed possible implementation where all the devices are simple HTTP servers. Different implementations were proposed, depending of design choices, without any change on the side of the devices. Hence a future work could be to study the specification and implementation of variants depending on non functional needs.

REFERENCES

[1] S. Tilak, N. Abu-Ghazaleh, and W. Heinzelman, "A taxonomy of sensor network," *Mobile Computing and Communication Review*, vol. 6, no. 2, 04 2002.

[2] L. Reveilleau, E. Becquet, L. Bacon, J.-M. Douin, E. Gressier-Soudan, and F. Horn, "Towards a rt-java based embedded remote monitoring tool for small and medium power plant units." Proceedings Emerging Technologies and Factory Automation (ETFA'2001), 10 2001.

[3] M. T. Jones, "An embeddable http server," *Dr. Dobb's Journal*, 10 2001.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[5] R. D. Giorgio, "Serve clients' specifi c protocol requirements with brazil, part 1-4," *JavaWorld*, 08 2000. [Online]. Available: www.javaworld.com/javaworld/jw-08-2000/jw-0811-javadev˙p.html

[6] M. Hauswirth and M. Jazayeri, "A component and communication model for push systems," in *ESEC/FSE 99 - Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, 1999, pp. 20–38.

[7] P. T. Eugster, R. Guerraoui, and J. Sventek, "Distributed asynchronous collections: Abstractions for publish/subscribe interaction," in *14th European Conference on Object Oriented Programming (ECOOP 2000)*, 06 2000, pp. 252–276.

[8] E. Cariou, A. Beugnard, and J. Jezequel, "An architecture and a process for implementing distributed collaborations, 6th international enterprise distributed object computing conference (edoc 2002), 17-20 september 2002, lausanne, switzerland, proceedings," in *EDOC*. IEEE Computer Society, 2002.

[9] A. Beugnard. Communication components. [Online]. Available: http://www-info.enst-bretagne.fr/medium/index.html

[10] ibutton temperature sensor. [Online]. Available: http://www.ibutton.com/products/ibuttons.html#temperature

[11] R. D. Giorgio, "An introduction to the url programming interface," *JavaWorld*, 08 1999. [Online]. Available: www.javaworld.com/javaworld/jw-08-1999/jw-09-javadev.html

[12] R. Englander, *Java and SOAP*. O'Reilly, 2002.

[13] D. Cox, "Xml for instrument control and monitoring," *Dr. Dobb's Embedded Systems*, 11 2001. [Online]. Available: http://www.ddjembedded.com/resources/articles/2001/0111i/0111i.htm

[14] J. Breed. (2000, 08) Instrument markup language website. [Online]. Available: http://pioneer.gsfc.nasa.gov/public/iml/

[15] J. Waldo and K. Arnold, *The Jini Specifications*, 2nd ed. Addison-Wesley, 11 2002.

[16] Brazil documentation and api. [Online]. Available: www.sun.com/research/brazil/

[17] D. Loomis, *The TINI specification and developer's guide*. Addison Wesley, 2001. [Online]. Available: http://www.ibutton.com/TINI/book.html

[18] J. Elosua and J. Burgillo, "Www-based remote cotrol using tini cards and brazil." IFAC 15th Triennal World Congress, Barcelona, Spain, 2002.