72975 - Software Systems Engineering Project Work M

Final Report

Matteo Olivi
matteo.olivi2@studio.unibo.it
Aprile 2018

Table of contents

1. A broad overview of the problem

The area or research of this project work is that of the IoT, that is, that of a world where physical devices carry electronic equipment on them (processors, sensors, actuators) and are capable of interacting with each other at a global scale through the Internet.

Of course, the glue that will keep all of these devices connected together is Software implementing interaction/communication protocols. The problem which is not solved yet is that of the interaction between devices that run different software and/or protocols, for instance because they are produced by different vendors. What happens is that device A can be accessed only through protocol A, whereas device B only speaks protocol B, thus they cannot talk to each other. Currently there's a plethora of different protocols used in IoT, depending on the use-case of the device, on the producer and on other factors. Since it is very unlikely that this will change in the future, it becomes crucial to find ways to enable communication between software components with different interfaces and speaking different protocols.

The most straightforward solution to this problem is to add a layer of software between the two endpoints wishing to interact, and have this intermediate layer act as an interpreter between them. By wrapping a piece of software PSA that only speaks protocol A (e.g. Zigbee) with an interpreter from protocol B (e.g. REST) to A and vice versa, it becomes possible for PSA to communicate with another piece of software PSB that only speaks protocol B. Notice that logically there needs to be such an interpreter for every protocol other than A that we want to support. Also notice that the process of wrapping legacy software with an interpreter is potentially infinite. For instance, assume you have a lamp which can be turned on and switched off by invoking a POJO's methods. By itself this is not very useful because we want to be able to act on the lamp remotely when we are near it. Thus, we can add to the lamp another software layer, that will enable communication with the lamp over Zigbee, solving our problem. More precisely, this interpreter will "listen" to commands issued by a remote user over Zigbee to toggle the lamp state and turn them into method invocations on the POJO controlling the lamp. We have just added an interpreter. What if now our requirements change, and we need to support interaction with the lamp from locations which are far away from it as well. Zigbee is a PAN protocol, thus it doesn't enable long range communication. At this point, we need another software layer acting as an interpreter, one that is capable of enabling long-range communication, for instance a REST server. Also assume that while the Zigbee-to-POJO interpreter was deployed on the lamp, there's not enough memory left to do that again. Thus, our REST server will have to be deployed close to the lamp but on a different device (e.g. a Raspberry Pi), and instead of translating from REST to method invocations on the POJO, it will

translate from REST to Zigbee. Then, the Zigbee interpreter will turn the received request (initially in REST format) into a method invocation on the POJO. We can see that we can keep adding software layers indefinitely to enable interoperability. Therefore, finding a flexible, robust and elegant solution to this interoperability problem is of paramount importance.

Finally, notice how this interoperability problem appears multiple times at different layers. In the previous paragraphs, the term “protocol” has been used vaguely on purpose. In fact, everything we have said is true both for application layer and lower layers protocols. To clarify this, imagine that device A and device B want to interact. If A uses TCP at the transport layer, whereas device B uses UDP, no communication is possible, thus we need an interpreter. But we are not done yet. Even with our interpreter, or assuming that A and B both speak TCP (UDP), for meaningful communication to take place the devices must share some application level protocol, such as the semantics of the exchanged data. If A is the remote lamp of the previous example and only accepts as commands the strings “ON” and “OFF” whereas B can only issue the strings “LIGHT” and “DARKNESS”, we need an additional interpreter at the application layer (or A and B to agree on an application layer protocol/data semantics).

2. Our starting point

With respect to the problem described in Ch. 01, we didn't start from scratch. In fact, our starting point was the QActor metamodel, whose purpose is to develop models (description of structure, interaction and behavior) of distributed systems. Its core abstraction is that of a QActor, which is similar, but not identical to, an AKKA actor (the Q in the name stands for "quasi"). QActors run on computational nodes called Contexts (identified by a name, an IP address and a port number), and interact through messages and events. In the QActor metamodel, both messages and events have a precise syntax, and at the implementation level are both exchanged as strings (because strings are basically the same thing in every language, this enhances interoperability). What differs is their semantics. A message is sent from a sender to a specific recipient (another QActor in the QActor metamodel) and is buffered (i.e. it's never lost and it can be processed by its recipient long after it was delivered by the underlying infrastructure). An event is emitted by an emitter, has no specific recipient but can instead be received by every QActor in the system, or belonging to systems linked to the one where it was emitted (more on this later). Events are not buffered: when an event is emitted, QActors that were not ready to "catch" that event will miss it and have no way to process it. Finally, QActors are message/event-based rather than message/event-driven. In a nutshell, this means that they always choose explicitly when to process an event or message, or, from the opposite perspective, there's no way an event or message can force some behavior in a QActor without the QActor explicit intention to react to that event or message.

Why is the QActor metamodel relevant with respect to the interoperability problem described in Ch. 1? Because it provides a partial solution to that problem. In fact, it provides a way for different systems to agree on an application layer protocol. First, in a system (or more systems working together) of QActors interaction happens entirely through messages and events. Second, when defining a QActor system, all the messages and events used must be explicitly listed (names, syntax, etc…). Together, these two pieces of information make up a complete application level protocol.

In general, when we describe a system through a QActor model, we have to specify all the QActors involved, the Contexts where they will run and the messages and events that will be exchanged in the system. For our problem, this is a severe limitation: in the IoT application domain we don't know in advance which devices will be there, some might go away, other might be added later, etc… . For instance, if we start with our Zigbee lamp, we start with a system where there's only a QActor modeling the lamp. If later we want to add our REST server to enable long distance control over the lamp, we can map it to another QActor, but we have to

modify the initial system model where the lamp is defined by adding this new QActor, which is really inconvenient. The ideal mechanism we are looking for should let us add/remove new devices/entities (mapped to QActors) without touching what was already there, as this suits the dynamicity of the IoT applicative domain better. As a matter of fact, the QActor metamodel already provides that: the standalone flag. When we define a QActor system, we can reference a Context which is already running independently and which was defined in a different QActor system by using the standalone flag (the details on how to do that are out of the scope of this document and are left to the reader, they can be found in the QActor doc). With respect to our example with the Zigbee lamp, this means that we can start by defining the QActor system where the only QActor is the one modeling the lamp. If later we want to add the QActor modeling the REST server, we can do that in a new, separate QActor system, without touching the one with the lamp. There are some constraints: the new QActor system will have to use the standalone flag to reference the Context where the lamp runs (which means it will have to know its name), it will have to declare some of the messages and events that the lamp uses, and, provided that it wants to use messages to communicate with the lamp, it will also have to know the name of the QActor modeling it (because of the semantics of messages). Notice that what the two QActor systems are doing by using the same messages and events is agreeing to use the same application level protocol. This is one of the two fundamental steps described in Ch. 01 to achieve interoperability: sharing an application level protocol, and sharing lower-level communication protocols. The QActor ecosystem also solves the second problem. In fact, it comprises a Software factory that automatically converts a QActor model to Java code. This allows to go from a high-level model of the system to a close-to-runnable implementation of the system. The specific business logic code of the application is still responsibility of the application developer, but all the boilerplate code for the communication of the QActors and other “mundane” tasks is generated by the software factory. This means that it is the software factory that automatically picks the lower (with respect to the application layer) layers protocols and takes care of communication at those layers. To sum it up, the QActor ecosytem (metamodel, software factory) solves both of the issues of application level and lower layers communication /interoperability by defining (or by forcing the user to define) and using certain communication protocols. What’s more, it allows to put together different systems in a modular and flexible way (through the usage of the standalone flag). It looks like the interoperability problem in IoT is completely solved by QActors. Of course, this is not the case.

To really have interoperability we need all of the three aforementioned advantages: common application layer protocol, common lower layers protocols, support for heterogeneity. The QActor toolkit gives you the first two provided that your system is entirely made by QActors, that is to say, that your system is homogeneous, but that is too much of a constraint, and as mentioned in Ch. 01, one of the things that makes interoperability in IoT hard is the plethora of different languages/protocols/technologies used! While using QActor provides many

advantages, assuming that everyone is using them or that everything is built with them is not realistic. Thus, the main focus of this project work has been on heterogeneous systems (QActor to non-QActor) interoperability, and it has been split into two halves: the first one focusing on application level interoperability, ignoring interoperability at lower layers. The second half has been the exact opposite: we forgot about application layer interoperability and focused on lower level mechanisms to allow communication.

3. QActor to non-QActor interaction: application level

NOTE: In this chapter the adjective “external” is used to identify a software component which is not a QActor and was not modeled in a QActor system, but independently.

As mentioned in Ch. 02, not all the devices/systems will be modeled as QActors. This means that for the QActor ecosystem/toolkit to be effective, it must be easy to integrate it with systems not modeled as QActors. For instance, assume the QActor factory only produces code in language X. Maybe there’s a component/device in our system that, because of other requirements, must be written/modeled in language Y. While we can still model this component as a QActor, using the QActor software factory is not an option, which means that the application developer has to take care (at least partially) of communication between different parts of the system. Some small proof of concept examples were developed during this part of the project work to show the flexibility of QActors even in these scenarios. The point was showing that it’s easy to make QActors speak to non-QActor entities at an application layer. To not bother with lower layers communication, we assumed that QActors and non-QActor software used the same protocols. We achieved this by looking at the protocols used by the QActor runtime implementation and using the exact same protocols for the non-QActor software (e.g. external components, etc…). As a matter of fact, this protocol was TCP, thus we used TCP Sockets. As for the language used for non-QActor software, we picked Javascript on Node.js, since we believe that it’s features (e.g. single-thread and asynchronous IO) make it suitable for IoT. Notice that any language choice would have worked though, and that now the QActor software factory also partially supports Node.js.

So how did we tackle the issue of application level interoperability? First of all, notice that QActors only use messages and events. From an implementation point of view, these are strings with a specific format (declared in the QActor system) sent over whatever underlying transport protocol is used by the QActor runtime (currently it’s mostly TCP). Since strings are supported in every language, all an external process wishing to communicate with a QActor system has to do is connecting with a TCP socket (or whatever transport protocol is used by the QActor runtime) to the QActor runtime in the systems/contexts where the recipient QActors are and send over that socket a string whose format is compliant with that of the message or event it wishes to send. The QActor runtime will handle the delivery with the desired semantics (message/event) to the affected QActors. Of course, this requires the external entity to know how to reach out to the QActor recipient: IP and port of the Context where it runs, name of the QActor and name and format of the messages/events to be used. Notice once again that interoperability at the application layer is straightforward even in this case: the only requirement

is using strings whose formats match that of messages and events defined at the top of the QActor system.

As for the interaction in the other way, that is, from QActor to non-QActor, things are trickier. The QActor runtime does not support delivery of messages/events outside of the QActor system, which means that some custom logic within that QActor system will have to be implemented by the application developer, for instance by using the QActor metamodel *actorOp* keyword to define custom behavior of a QActor. As an example, a method that directly sends data to the external entity through a TCP socket might be implemented, or one that loads the data into a shared database, etc… .

3.1. Completely separate systems interaction

The first scenario we thought about is that of a QActor system already up and running. Then there's the need to make it interact with an external component, which has to be developed in a different language (Node.js). This matches real world scenarios where a new part of the system must be integrated a posteriori and the same language/technology used for the previous part of the system can no longer be used. Using once again the lamp example, we have our Zigbee lamp modeled as a QActor, and later we want to expose it to the world using a REST server, but we can't use QActors anymore (because of other requirements).

The existing QActor system knows nothing about the new external system, all it does is defining the events and messages that need to be used to interact with it. More specifically, in our running demo for this case there's a QActor system with a single QActor defining a message to interact with it, which simply waits for that message. The model is in Fig. 1.1. nn the next page.

Then there's a Node.js script that sends a string with the same format of the QActor message over a TCP socket connected to the QActor runtime running in the same Context as the QActor. The QActor receives the message correctly, even if it was not created in a QActor system.

The code for this example and everything needed to run it and build it can be found here.

```
System demo1

Dispatch test_msg : test_msg(x)

Context  ctx ip [ host="localhost" port=8000 ]

QActor receiver context ctx -g yellow {
        Plan wait_msg normal
                actions [
                        println("receiver waiting for a message")
                ]
                transition whenTime 100000000 -> wait_msg,
                                whenMsg test_msg -> handle_msg
        Plan handle_msg
                [
                        onMsg test_msg : test_msg(x) -> printCurrentMessage
                ]
                switchTo wait_msg
}
```

Fig. 1.1

3.2. Heterogeneous yet reciprocally aware systems interaction

This example covers the use case where all the components are known in advance, but you can't use QActors for some of them. Following along the lamp example, we know that a REST server is needed together with the Zigbee lamp from the beginning, but because of some other requirement we can't use Java for the REST server, thus the QActor software factory cannot be used (of course, we can still model the server as a QActor because models are intrinsically useful, even without a software factory). In this system, there's still a receiver QActor that does nothing but waiting for a message. Notice that in the example in 3.1. the start up of the external sender must be done manually (or with a tool different from those offered by the QActor system) because the QActor system knows nothing about it. Now it's different, while the sender is not a QActor, we know from time 0 that it's there, therefore it makes sense to embed this knowledge in the QActor system to make things easier: the system will feature a sender QActor as well, which will make nothing but starting the Node.js script implementing the actual sender. Other than that, the sender is the same as the one in section 3.1. The QActor model can be found in Fig 1.2 in the next page.

A link to the repository with all the code and the necessary to build and run the demo is here. There's also a repository here with the very same example but enhanced from a Software

engineering perspective (the sender is not just a single Node.js script, but its functionality is split into separate Javascript files for better reusability and separation of concerns).

```
System demo2

Dispatch test_msg : test_msg(x)

Context ctx ip [ host="localhost" port=8000 ]

QActor receiver context ctx -g yellow {
        Plan wait_msg normal
                actions [
                        println("receiver waiting for a message")
                ]
                transition whenTime 100000000 -> wait_msg,
                                whenMsg test_msg -> handle_msg
        Plan handle_msg
                [
                        onMsg test_msg : test_msg(x) -> printCurrentMessage
                ]
                switchTo wait_msg
}

// It's called "sender_0" instead of "sender" only because "sender" is a reserved keyword
QActor sender_0 context ctx -g cyan {
        Plan init normal
                actions [
                        println("sender up and running")
                ]
                switchTo send_msg
        Plan send_msg
                actions [
                        actorOp sendFromNodejs
                ]
}
```

Fig. 1.2

3.3. A standalone executor

In this example we experimented with the idea of having a standalone executor which does no predefined task, but waits for tasks to be submitted to it externally (as source code). We believe this could be useful in dynamic and rapidly changing scenarios like those of IoT applications.

To better understand this, assume you have a QActor System implemented in Java. Now assume that the QActor system must be integrated with some additional functionality, and the source code that implements it is already available but in a different language, say Javascript, as a script. The problem is, for some reason we cannot install a Javascript interpreter on the nodes where the QActors run. We could implement manually that functionality as a QActor, but that wouldn't be efficient since it is already implemented. The idea is to have on a different node an "executor server", more specifically a Node.js server, that waits for incoming tasks to execute.

This executor knows nothing about QActors and the tasks that will be submitted to it, thus it's flexible and independent from its users. Notice that as a consequence, if the QActor system expects some results back from the task available as a Javascript script, the sending of such results (as messages and/or events) back to the QActor system must be implemented in the script itself: we can't rely on the executor doing that autonomously, as this would tightly couple it both to the QActor system and to the specific task being submitted, thus making it non-reusable.

More specifically, in the proof of concept implementation, the executor is an independent Node.js script which waits on a TCP socket for incoming Javascript code to be executed. Notice that using Javascript makes things easier because of the eval function, which allows us to dynamically execute code. Then, there's a QActor which reads the Javascript file from the file system and sends its content (i.e. Javascript code) as a String over a TCP socket connected to the executor. Notice that the submitted job produces a result that the sender is interested in, thus the Javascript code also contains the sending of that result back to the QActor system. Notice that to fully leverage the QActor facility and to have a higher degree of abstraction, these results are not sent back directly to the sender, but as messages/events to the QActor runtime, which will deliver them to the sender normally.

The code for this example can be found here. The QActor model is:

```
System demo4

Dispatch result : result(payload)

Context ctx ip [ host="localhost" port=8000]

QActor client context ctx -g yellow {
        Plan transmission normal
                actions [
                        println("Client about to submit job");
                        actorOp submitJob("./srcJS/customJobs/getTemperature.js");
                        println("Client submitted job")
                ]
                switchTo wait_result
        Plan wait_result
                actions [
                        println("Client waiting for result")
                ]
                transition whenTime 20000 -> collect_result
                                whenMsg result -> collect_result
        Plan collect_result
                [
                        printCurrentMessage
                ]
}
```

Fig. 1.3

Finally, notice that this approach has some limitations. First, the executor executes code received from other entities: this is in general a poor thing as for security, some mechanisms to avoid execution of (either intentionally or unintentionally) harmful code should be used, adding overhead. Second, the submitted code might depend on libraries. We must make sure that such libraries are installed on the executor as well, or that the executor is capable of installing them as needed. One solution to both these problems is that of implementing the executor as a container runtime, and instead of submitting plain source code, a container image could be sent.

4. Interaction at layers below the application layer

In the second and last part of this project work, we thought about low level (i.e. below the application layer) interaction. More specifically, we focused on IPC mechanisms, that is, we tackled the problem of intra-node communication. Assume again we have our lamp, accessible through a POJO. Now assume that we deploy on it a REST server to make it accessible from remote. There must be an interpreter translating REST commands to method invocations on the lamp POJO. Notice that this interaction is intra-node, and that in general, to have flexibility, we would like it to take place regardless of the languages the REST server and the POJO are written in. In a nutshell, we would like to find an interaction/communication mechanism optimized for intra-node communication that allows communication of heterogeneous (i.e. written in different languages) components. The use case for this is whenever we need to make a plug-in for a physical IoT device that allows to interact with it locally, but in the future we might have to make it accessible from remote: what interaction mechanism with the plugin makes it possible to perform this addition as quickly as possible?

The most straightforward solution is that of using TCP/UDP sockets. They are a standard: the socket API is available in every language, thus heterogeneity is completely taken care of. The problem is, they are meant to be used for remote communication. While this is good for flexibility, since they work both for intra-node and inter-node communication (thus allowing the two endpoints to be deployed on different nodes), it is far from optimal for the intra-node case: data being sent has to go through the whole TCP/IP protocol stack twice, which is really bad for performance.

A solution that only works for intra-node communication is that of using files to communicate: the sender writes its message/command with a predefined format on a file and the reader (our legacy plugin) reads it. More specifically, the reader could register an observer on the file to avoid polling it for new commands/new data. There are multiple libraries that allow that (e.g. NIO in Java). The problem with this solution is that of performance once again: it requires

disk IO twice: the first time to write data, the second to read it. Disk IO is a notorious performance bottleneck, thus this is not an improvement with respect to TCP/UDP sockets, which might even be faster (of course, for more authoritative and accurate comparison, a performance evaluation should be carried out, which is out of the scope of this project work).

Since the two interacting parties are on the same node, the general guideline to make their communication as efficient as possible is to leverage memory on that node: they can use the same memory to communicate. This has neither the disk IO performance bottleneck, nor the overhead associated with going through the TCP/IP protocol stack. Thus, we asked ourselves what IPC mechanisms that only used memory were already available. For our analysis we began by focusing on Linux OS, as IPC facilities vary across different OSs. More specifically, we were already familiar with two Linux IPC mechanisms, both relying only on memory: Named Pipes (FIFOs) and Unix Domain Sockets.

4.1. Using named pipes

A named pipe, or FIFO, is a kernel facility with a name in the file system hierarchy, so that it is accessible for anyone who knows its name. It's implemented as a pipe, and it's mostly used in the same way. More specifically, it is a unidirectional (one way) data channel: there are two ends, one for reading and the other for writing. The sender writes data with a write call (blocking if the buffer is full) on the writing end, and the receiver reads data with a blocking read call on the reading end. All the data flowing through a FIFO (pipe) is buffered in a buffer in the kernel memory space. This means that when two processes A (sender) and B (receiver) communicate over a FIFO, data is first copied from A's memory space to the kernel's buffer backing the FIFO (send), and then from the kernel's buffer from B's memory space. This is an improvement with respect to TCP/IP sockets and files as IPC mechanisms: only memory is used, thus it is faster. Notice though that memory is not shared between sender and receiver, neither copied directly between the sender's buffer and the receiver's buffer, but mediated by the kernel's buffer. This is an overhead which could possibly be optimized (i.e. at least theoretically it should be possible to outperform FIFOs by using real shared memory). Another thing worth mentioning about FIFOs is that message boundaries are not preserved: if multiple senders send data on the same FIFO, if n messages are written before a read is performed and all fit into the buffer they will be read as a single "chunk". Also, for writes of data bigger than the FIFO's buffer, interleaving is possible. Thus FIFOs add the burden of parsing messages at the receiver's end.

To experiment with FIFOs as an IPC mechanism, we developed a small proof of concept example, where a program written in Java reads a message sent from a Node.js script. Of course,

this communication takes place through a FIFO. From the Java end, the FIFO can be accessed like a normal file with the Java.io API (i.e. the Java is not aware that the FIFO is not a “normal” file). From the Node.js end, the FIFO can still be accessed as a normal file with the ‘fs’ module API. The complete code of the demo can be found here.

4.2. Using Unix Domain Sockets

The other IPC mechanism we already knew was that of Unix Domain Sockets (UDS). A UDS can be accessed through the TCP/UDP socket API. As a matter of fact, they are semantically equivalent to regular TCP/UDP sockets. What changes is that UDS can only be used for intra-node communication, and are therefore optimized for that: data will not flow through the TCP/IP stack, but will only be copied to and from the kernel buffers (in a fashion similar to FIFOs). Also, UDS are bound to local file system names, not to TCP/UDP ports. As we have already mentioned, the underlying implementation is similar but not identical to that of FIFOs: data is copied from a sender’s buffer to a receiver’s buffer (maintained by the kernel). Notice though that UDS, unlike FIFOs, are bidirectional, and if more senders want to talk to the same receiver, they’ll get separate data channels (UDS), while the receiver will have a UDS to accept incoming connection requests and an additional UDS for actual data exchange for each ongoing connection (just like with TCP/UDP sockets). The fact that multiple clients get their separate UDS implies that no interleaving between their messages can happen, which in turn means that there’s less overhead at the receiver’s end to parse and sort data received from multiple senders. This in general is an advantage with respect to FIFOs. Notice though that our specific use case is that where the FIFO/UDS is used to communicate with a legacy plugin by a single entity (e.g. a REST server) receiving requests by remote clients, which means that there will be only a sender. Thus, the aforementioned advantage is not paramount (albeit it could still be nice to have if the sender was multithreaded, because each thread could act as a separate sender).

One big disadvantage of UDSs is that they can only be created and accessed through native calls. This means that accessing them from languages other than C or Bash has a considerable overhead. For instance, in our example we wrote the server in Java. There’s no way to use a UDS from Java which does not involve using JNI (or JNA), which is not straightforward. More specifically, we didn’t use JNI directly, but used a library whose specific purpose is that of providing a wrapper for UDSs in Java (of course, that library was implemented using JNI). Thus, if you don’t want to use JNI yourself, you could rely on a third-party library. This can be risky because the library could stop being maintained. Also, if the library enforces a specific conceptual model, you’re bound to that model. For instance, the library we used only allowed to implement the server as a single thread using a selector and non-blocking IO (in a Java.NIO fashion). If we had wanted to use the traditional approach (blocking-IO, one thread to

accept connections, one thread for each open connection), it wouldn't have been possible (with the library we picked).

To test the usability of UDSs for IPC, we developed a small Proof of Concept demo, where a Java server listens for incoming connections and a group of senders (written in Javascript over Node.js) send data to it. The complete example can be found here. At the Javascript end, the UDS socket can be accessed through the Node.js 'net' module. Notice that the sender must know the address of the server (i.e. the file name of the UDS used to accept connections).

4.3. Conclusions and follow-ups

While both FIFOs and UDSs are a fast mechanism for IPC (and for our specific use case), there's still room for improvement. On the one hand, they only rely on memory for communication: no disk IO, no flow through TCP/IP protocol stack. This is good. On the other hand there's no real shared memory: data is copied from one memory buffer to another multiple times. If the two communicating processes could share a memory segment for real, communication would be even faster. A follow-up to this work could be an investigation and evaluation of such mechanisms. Staying on a UNIX-like system, using shmget or mmap system calls could be a starting point to achieve that. Notice though that just like with UDSs it would be necessary to use native invocations, which is tricky to do in languages other than C or Bash (e.g. in Java we would have to wrap them with JNI or JNA). Alternately, Java NIO provides the class MappedByteBuffer which should make shared memory between processes written in different languages possible even without using JNI (and should also be more cross-platform). This is another thing to investigate.

As a final observation, we suggest a future improvement of the QActor metamodel and associated software factory. It would be beneficial if such software factory could use different low level communication mechanisms (not just TCP sockets). The rough idea is to provide an API through some interfaces that is the same for all the underlying communication mechanisms, and have the software factory implement this interfaces with the most suitable mechanisms each time. For instance, if all the Contexts are on the same physical node, the software factory can realize that autonomously and use intra-node IPC mechanisms (such as those described in this report) instead of TCP sockets to enable low-level communication. Notice that IPC mechanisms are typically not portable (e.g. each OS has its own mechanisms which are not available on other OSs, as mentioned before we focused on Linux), but this issue can be easily dealt with by having the Software Factory checking which OS is below and choosing a mechanism supported by that OS (as a design suggestion, the GoF Factory pattern would be ideal for this task). For those

choices where the software factory cannot decide autonomously, a flag in the QActor metamodel is the most straightforward way to give the user the possibility to instruct the software factory on what to choose.