FTTlet based distributed system architecture

Mário J. B. Calha
Polytechnic Institute of Castelo Branco - EST
Av. do Empresário
6000 Castelo Branco
mjc@est.ipcb.pt

José A. G. Fonseca
University of Aveiro – DET/IEETA
Campus Universitário de Santiago
3810-193 Aveiro, Portugal
jaf@det.ua.pt

Abstract

In this paper an architecture of a distributed embedded system, that offers the advantages of the Java programming model and the flexibility and timeliness of the Flexible Time-Triggered (FTT) paradigm, is presented. The architecture builds on previous work by the authors in which a simple mechanism to dispatch tasks and messages was proposed for CAN-based distributed systems. In order to overcome the innate limitations of the under-specified scheduler built into Java Virtual Machines (JVM) this proposal disables it and makes use of the centralized scheduler of the FTT approach. This way, one of the main constraints in using Java in distributed embedded systems is eliminated. Due to the use of an underlying kernel that isolates the JVM from the network and the FTT mechanism, any JVM implementation with real-time guarantees doesn't need significant changes in order to be integrated in this architecture. Also a framework that allows the complete system simulation is discussed.

1. Introduction

An embedded system integrates, basically, a processing unit, memory and I/O connections. Due to the always increasing computational power of embedded processors and significant memory sizes, it is now reasonable to expand the software layer to include a Java Virtual Machine (JVM).

One of Java's strengths is that it can run on any machine for which there is a compliant JVM. Every Java program, when it is compiled, gets translated into the same computer code. Every different hardware/operating system set has a program - the JVM - that can interpret between this universal Java computer code and the more case-specific parts of the software and hardware. This means that the exact same Java code can run on any computer system.

An open issue in these systems is still the timeliness guarantee. Due to the way the garbage collectors work it is not possible, in a standard Java system, to know the execution time of the programs, making it unusable for real-time applications. But an effort for the development of JVMs that can guarantee timeliness is going on. Currently, at least, two Real-time extensions for Java have been proposed. One is the "Experts Group Real-Time Specification for Java" (RTSJ) [2] and the other is the "J-Consortium Real-Time Core Extension" (RTCore) [3]. Implementations of these specifications are already available bringing the Java benefits to the real-time community.

In [5] the Distributed Real-Time Specification for Java (DRTSJ) introduces the Distributed Real-Time Remote Method Invocation (RMI) model. The DRTSJ is focused on supporting predictability of end-to-end timeliness for sequentially trans-node behaviours (e.g., chains of invocations) in dynamic distributed object systems. The integration and extension of the existing RTSJ and Java Remote Method Invocation (RMI) facility to provide the basis for the Distributed Real-Time Specification for Java (DRTSJ) is being investigated in [4] and [6].

The use of resource management strategies in supporting the testing and certification of real-time, fault-tolerant, mission critical systems based on distributed object middleware was explored in [8].

In previous works, [9] and [10], we have proposed a simple mechanism for distributed systems that allowed the dispatching of tasks and messages in a time-triggered manner using the Flexible Time-Triggered (FTT) paradigm. These works led to data flow analysis and to the derivation of requirements to the tasks and messages parameters. Also a simulator to preview the timeliness of the transmission of messages and of the execution of tasks in a distributed system has been proposed [12].

This paper is organized as follows, section 2 presents an architecture of a distributed embedded system, section 3 is an overview of the system operation, section 4 indicates an interesting application of this architecture in the form of a simulation framework and in section 5 some conclusions are drawn.

2. Architecture of a Distributed Embedded System

A typical distributed system is composed of several nodes interconnected by some network that can be wire-

based or wireless. The architecture presented here uses a Java Virtual Machine (JVM) running on top of a basic kernel. This JVM allows the execution of special Java programs, called FTTlets, which are executed upon the arrival of a triggering event from the FTT engine. See Figure 1.

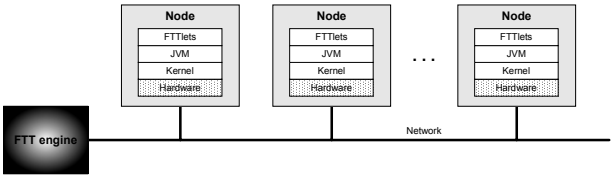

Figure 1. A distributed embedded system.

The FTT engine contains and manages FTTlets through their lifecycle. The FTT engine takes care of:

- FTTlet admission control;
- Scheduling;
- EC Trigger Message generation, this dispatches the FTTlets at each node and orders the transmission of messages;
- Kernel, JVM and FTTlet loading (not mandatory).

Currently this architecture can be implemented using the FTT-CAN protocol, [1] and [11], or the FTT-Ethernet protocol [7] to trigger the dispatching.

2.1. Kernel

The kernel offers services to the JVM in the form of an Application Programming Interface (API) and services to the FTT Engine in the form of a System Programming Interface (SPI). These services mainly include inter-node task communications (message exchanging) and FTT engine event decoding.

The kernel is organized in several layers as depicted in Figure 2.

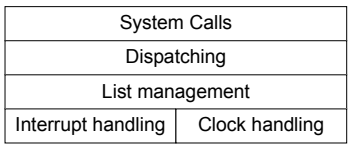

Figure 2. Kernel layers.

The functions of the layers are:

- System Calls – services provided by the kernel to the FTT engine and the JVM;
- Dispatching – upon reception of an event from the FTT engine, this layer instructs the JVM about the next FTTlet to be dispatched;
- List management, interrupt handling and clock handling – basic kernel functionality.

The absence of a scheduler is justified by the centralized scheduling mechanism of the FTT.

The use of a kernel instead of an adapted JVM that handled directly the hardware, the network and the FTT implementation, has great benefits. For each architecture there has to be a specific implementation of the kernel but the JVM only needs to be recompiled.

2.2. Java Virtual Machine

The JVM is the platform of execution of the FTTlets and has to handle the intra-node FTTlet communication.

The API is made available in the form of packages. These packages include the standard Java classes (depending on the edition) and system specific packages that allow operations like Sending and Receiving messages (FTT package) and accessing the hardware features.

Due to the constant evolution of the JVMs, especially in what concerns real-time, the JVM is running on top of a kernel. This approach isolates the JVM from the underlying architecture. To accomplish this isolation the JVM has to be coupled with a set of native functions that interface it with the kernel.

The under-specified thread scheduler, an issue in the use of Java in real-time systems, is disabled. Just like in the kernel, the scheduling is handled by the centralized scheduling mechanism of the FTT.

The thread dispatching unit of the JVM is also disabled. So the dispatching is handled by the node kernel, upon reception of the trigger event.

2.3. FTTlet

Due to the use of the FTT paradigm, the FTTlets are Java programs (resembling servlets) that are started through the network, i.e. are executed upon arrival of a triggering event. This event is caught by the local kernel. The event contains information about the messages and tasks that should be dispatched. If there are messages to be dispatched the local kernel starts their transmission. If there are FTTlets selected for dispatching the local JVM is instructed to execute them. The FTTlets are loaded into each node, by the FTT engine, according to the requirements.

A FTTlet is a Java technology based component, managed by a FTT engine. Like other Java-based components, FTTlets are platform independent Java classes that are compiled to platform neutral bytecodes that can be loaded dynamically into any node and dispatched by a Java enabled FTT engine.

The key benefits of the FTTlets are:

- FTTlets are fast because they are loaded into memory once, and run from memory thereafter;
- FTTlets are relatively simple to implement;
- Since FTTlets are Java byte code, they are platform independent by nature.

3. System operation

3.1. FTT package

In order to offer the functions needed by the FTTlets to interoperate with the rest of the system a FTT package

is available. This package provides a standard way to access the FTT features of the system.

The FTT package is based on the FTT class. This class offers the following methods:

- ReadMessage
- PostMessage

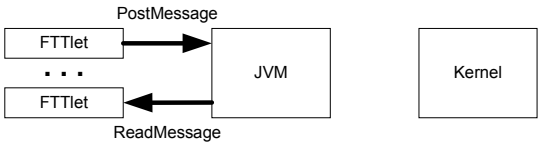

Figure 3. Message transaction in the same node.

If the message transaction is with a task running in the same JVM this package handles the transaction (see Figure 3). If the message transaction is with a different task then these methods interact with the kernel exchanging data between it and the FTTlets running in the JVM.

3.2. Kernel

The kernel is the cornerstone of the system because it allows the JVM and FTTlets to execute without being aware of the underlying system configuration. The kernel has to be adapted to each possible situation that depends on:

- The network topology,
- The transport protocol,
- If there is a real, or simulated, system.

The kernel offers two groups of services, namely: the application programming interface (API) and the system programming interface (SPI).

The API is a set of functions that are available to the JVM. These include:

- ReadMessage
- PostMessage

The method ReadMessage, is a blocking function, and is used to read a message sent by another task (or FTTlet).

The method PostMessage, is a non-blocking function, that is used to post a message to another FTTlet in a different node (see Figure 4). Therefore, the message interchange is the global communication paradigm that is used. One advantage of this model is the possibility of changing the node of execution of a task without interfering with the other tasks.

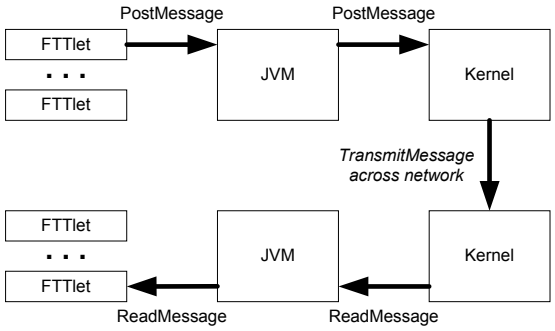

Figure 4. Message transaction between different nodes.

The SPI is a set of functions that are available to the FTT engine. These include:

- LoadJVM
- StartJVM
- LoadFTTlet
- StartFTTlet

The LoadJVM is used in the initialization procedure, where the FTT engine has to cooperate with the kernel for the loading of the JVM to the node's address space. The StartJVM is used to start the system. FTTlets can be loaded in a node through LoadFTTlet. The function StartFTTlet is basically an internal function that is called whenever the triggering event dispatches any FTTlet.

4. System simulation

One of the main benefits of this architecture is the possibility of developing and testing, transparently, the FTTlets in a simulated environment using a regular computer. This environment is supported by a special local kernel that is able to launch one JVM for each node of the simulation (Figure 5). This local kernel simulates the triggering events and takes care of the communication. Each JVM is unaware of the environment bellow.

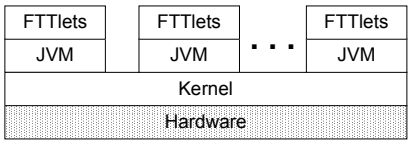

Figure 5. Simulation environment.

This approach facilitates the load balancing when node independent FTTlets are being executed.

Also, the system designer can make use of the simulation results in order to allocate features to each node, so that the nodes more heavily used can have some functionality moved to a different node.

4. Conclusions

A distributed system architecture based on the FTT paradigm was presented. This architecture offers a framework where each node has a Java Virtual Machine (JVM), running on top of a simple kernel, which allows the execution of FTTlets. The FTTlets associate the benefits of the Java programming language to the flexibility and timeliness of the FTT paradigm. This architecture can accommodate different network types and JVMs.

The possibility of a system simulation through the use of a different kernel was also exposed. This simulation can help the system designer: to achieve a better load balancing at each node and also to allocate the node specific features.

A demonstrator of this architecture is under development.

References

[1] Almeida, L., *Flexibility and Timeliness in Fieldbus-based Real-Time Systems*, PhD thesis, University of Aveiro, Aveiro, Portugal, 1999.

[2] The Real-Time for Java Expert Group, *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[3] J Consortium, *Real-Time Core Extensions*, 2000.

[4] Wellings, A., Clark, R., Jensen, D. and Wells, D., "Towards a Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation", The 22nd IEEE Real-Time Systems Symposium (RTSS 2001), London, UK, Dec/2001

[5] Clark, R., Jensen, E., Wellings, A. and Weels, D., "The Distributed Real-Time Specification for Java: A Status Report", Embedded systems conference, San Francisco, USA, Mar/2002.

[6] Wellings, A., Clark, R., Jensen, D. and Wells, D., "A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation", Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington D.C., USA, Apr/2002.

[7] Pedreiras, P., L. Almeida and P. Gai, "The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency", Proceedings of the 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, Jun/2002.

[8] Wells, D., Bernstein, R. and Vadlamudi, A., "Testability of Complex, Middleware-Based Systems," – Workshop on Dependable Middleware-Based Systems, Bethesda, Maryland, USA, June 2002.

[9] Calha, M.J., J.A. Fonseca, "Adapting FTT-CAN for the joint dispatching of tasks and messages", Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS'02), Vesteras, Sweden, Aug/2002.

[10] Fonseca, J.A., J. Ferreira, M. Calha, P. Pedreiras and L. Almeida, "Issues on Task Dispatching and Master Replication in FTT-CAN", Proceedings of the IEEE – AFRICON 2002, George, South Africa, Oct/2002.

[11] Almeida, L., P. Pedreiras and J.A. Fonseca, "The FTT-CAN protocol: Why and How", IEEE Transactions on Industrial Electronics – special edition on Factory Communication Systems, Dec/2002.

[12] Calha, M.J., J.A. Fonseca, "SIMHOL – A graphical simulator for the joint scheduling of messages and tasks in distributed embedded systems", To appear in the Proceedings of the FeT'2003 – 5th IFAC International Conference on Fieldbus Systems and their Applications, Aveiro, Portugal, Jul/2003.