Estimation of Bus Performance for a Tuplespace in an Embedded Architecture

Nicola Drago [#] Franco Fummi [#] Marco Monguzzi [‡]
Giovanni Perbellini [§] Massimo Poncino [#]

[#] Università di Verona
Dip. Informatica
Verona, ITALY

[‡] Sitek S.p.A.
S.Giovanni Lupatoto
Verona, ITALY

[§] Scientific Parc Verona
Embedded Systems Design Center
Verona, ITALY

Abstract

This paper describes a design methodology for the estimation of bus performance of a tuplespace for factory automation. The need of a tuplespace is motivated by the characteristics of typical embedded architectures for factory automation.

We describe the features of a bus for embedded applications and the problem of estimating its performance, and present a rapid prototyping design methodology developed for a qualitative and quantitative estimation. The methodology is based on a mix of different modeling languages such as Java, C++, SystemC and Network Simulator2 (NS2). Its application allows to estimate the expected performance of the bus under design in relation to the developed tuplespace.

1 Introduction

Control applications for industrial automation traditionally follows a hierarchical model, i.e., with slave devices organized around a master controller. Actuators and sensors, for instance, are usually driven by control software running on the master device. This inherently centralized models, however, lacks of flexibility: it typically requires reprogramming the master controller logic and the network configuration when new devices must be added to the system. Moreover, the master controller is a centralized failure point so that special algorithms and recovery solutions have to be implemented whenever a hierarchical model is used for mission-critical application.

Nowadays, trends toward designing controls systems as networks of "smart" devices have pushed towards a distributed approach, as opposed to the traditional master-slaves architecture. Several are the reasons for this shift of paradigm. Extensibility of a system is an essential feature: in complex control systems, it is commonplace to implement new functionalities by adding new devices. Scalability of performance is also a must: several instances of the same device could be networked to achieve best performance for a specific function. Finally, system redundancy is often a requirement: for mission-critical applications we may need to replicate some of the networked devices, and need "smooth" algorithms to replace possible failure points with their backups.

The use the tuplespace model [1] to connect devices in a control system seems attractive but the available Java implementations ([2], [3]) are not suitable for embedded networks made of low level/cost devices: most of them do not support Java or they even do not rely on an operative system endowed with a standard TCP/IP stack, making unfeasible the adoption of well known frameworks.

In this work, we present a strategy for evaluating the interconnection of heterogeneous devices and control applications using a tuplespace ("JavaSpaces-like") application middleware over a low cost high speed serial link used as a substrate on which to internetwork. The tuplespace middleware provides a layer on which distributed applications for control can be deployed, that includes: a discovery mechanism for communicating entities, a common interface schema language and repository, and an asynchronous communication using a common data scheme (tuples).

The serial link (TpWIRE / 1-wire specification) provides an ad-hoc channel for data communication suitable for mid-bandwidth (up to 1Mbyte/s) interconnects in embedded systems. TpWIRE has been used in the Exor Theseus system ([4]) given its support to real time communications and the required low cost to interconnect devices that are neither "smart" nor powerful enough to implement other standard protocols.

As an application of the proposed methodology, we have evaluated the impact of the tuplespace communication middleware by testing the communication between two TpWIRE endpoints, with respect to some typical traffic profiles of a real-life environment.

2 Tuplespaces

The concepts of *tuplespace* was first developed as part of the Linda System at Yale University in the 1980s ([5], [6]). JavaSpaces ([2]), TSpaces ([3]) and Ruples ([7]) are all notable Java implementations of tuple spaces based on the Linda concepts.

Tuplespaces are often described as globally shared, associatively addressed memory space. The central concept of a tuplespace is a set of agents communicating with other agents by writing, reading and removing *tuples* (an ordered set of typed values) in a universally visible repository (tuplespace). What makes the tuplespace model appealing is that it replaces synchronous point to point communication with asynchronous (and anonymous) associatively addressed messaging, where the messages are kept in a persistent message store. The basic element of a tuplespace system is a tuple, which is simply a vector of typed values, or fields. Tuples are associatively addressed via matching with other tuples. A tuplespace is simply an unstructured collection of tuples. Most tuplespace implementations provide both blocking and nonblocking versions of primitives for writing, reading and removing a tuple from the space. Moreover, primitives to support the "subscribe" (declare the interest of an agent on some kind of tuples) and "notify" (callback to subscriber) paradigm are usually provided.

2.1 Applicability to factory automation

The concept of tuplespaces has several interesting applications in the context of factory automation:

- *Scalability of systems*. A tuplespace is a very robust and reliable mechanism for parallel and distributed applications. Since the communication is asynchronous and anonymous, a variable number of agents can work together on a task. Applications following the "producer/consumer" pattern may particularly benefit from this model. For instance, let us assume an heterogenous network where nodes with different performance are mixed. If we identify as "producers" a set of agents (e.g., low performance nodes with no FPU support) putting in the tuplespace a vector of data and requiring for their *Fast Fourier Transform* computation (the required service) and as "consumers" (e.g., high performance nodes with FPU support) a set of agents able to easily handle these requests, the overall system performance are clearly proportional to the number of consumers. In general, the system performance can be scaled to deploy cost/effecting embedded solutions for factory automation.

- *Support to system extensions*. Tuplespaces usually support a "service discovery" mechanism. Devices exporting a service ("consumers" in the previous example) do register themselves into the service discovery subsystem. On joining the tuplespace, devices that need to use a service ("producers" in the previous example) query the discovery subsystem to locate the service and employ it. In this way, the system can inherently support the dynamic addition/removal of components (extensions) without requiring centralized control (and consequently re-programming or re-configuring).

- *Fault tolerant systems*. Some applications need to guarantee tolerance to the failure of some of their components. As an example, let us consider an actuator controlling a device made redundant as in Figure 1. An algorithm for detecting the failure of an actuator and getting the backup operating could move along following steps:

  1. At system startup, the *control agent* puts a tuple in the space requiring an *actuator agent* to start. It waits to start the control loop until the tuple is removed from space.
  2. At system startup, the various *actuator agents* try to remove that tuple. Just one of them will succeed[1], and it will set its state to *operating*. The others will set their states to *backup*.
  3. The *operating actuator* starts to execute its program semantics. On each tick, it also writes a tuple into the space signaling its state (something like: *operating OK*).
  4. On each tick, the *backup actuator* tries to remove the state tuple written by its dual. If the operation fails, the recovery procedure begins: the *backup actuator* changes its state to *operating* and starts executing the actuator program semantics.

The tuplespace paradigm offers the significant advantage of an *abstraction* of the communication infrastructure. In heterogenous environments such as factory automation appliances where most of the devices adopt proprietary protocols and data format, programmers dealing with the design of the agent(s) responsible for the control loop can focus on handling tuples from/to the "controlled environment" (i.e. sensors, actuators, etc), rather than facing directly the details of the underlying communication.

Indeed, the use of tuplespaces does not come for free: the overhead of passing messages through the tuplespace may dramatically impact the overall performance wherever bandwidth is limited.

[1] In a tuplespace, the timestamp on each tuple determines a total order relation.

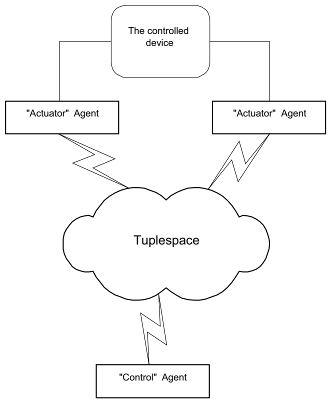

Figure 1. Redundant Actuators to Provide Backup in Case of Failure.

This raises the issue of developing an estimation methodology that allows to estimate the behavior of a link and the bandwidth required to support passing of latency-sensitive events (as in the case of factory automation appliances) through the tuplespace.

3 TpWire Architecture

This section describes the *TpWIRE* (Theseus Programmable Wires) bus and its potential scalability that will be investigated with the proposed rapid prototyping methodology.

3.1 Specification of 1-wire

A TpWIRE network is a daisy chain network with one Master and one or more Slaves as shown in Figure 2. The Master is responsible for initiating all communications over the network. Slaves can communicate with the Master only, but not between them. The Network requires one single-ended signal for the interconnection between network nodes (Master or Slaves) over short distances, while in the case of long distances a different signal is required.
A communication cycle begins by sending a TX frame

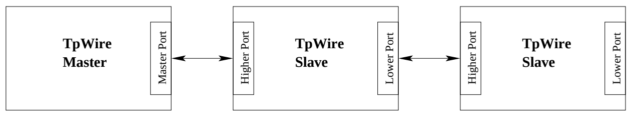

Figure 2. TpWIRE Network

from the Master towards all Slave nodes (Table 1). Only the selected Slave executes the TX frame command, and it replies by sending an RX frame to the Master. If any Slave responds within an expected time period, or an error occurs during the receive of TX or RX frames, the Master resends the TX frame a predetermined number of times before signaling an error. If a broadcast node is selected, all Slaves execute TX frame command and none of them replies. Each time a RX frame passes through the Slave (or a Slave generates it), the interrupt bit in RX frame is set if the Slave has a pending interrupt.

The TX frame consists of 16 bits; a start bit (always ’0’), three command bits CMD[2:0], eight data bits DATA[7:0] and four CRC bits CRC[3:0]. For write commands DATA[7:0] contains a valid data value, while for read commands it is ignored. CRC is calculated over CMD[2:0] and DATA[7:0] using the $x^4 + x + 1$ polynomial.

0	CMD[2:0]	DATA[7:0]	CRC[3:0]

Table 1. TX frame format

Each Slave automatically resets itself if no valid TX frame has been received within the Slave reset timeout period. Reset timeout period is fixed to 2048 bit periods of the currently programmed communication speed. When activated, reset stays active 33 bit periods.

A node ID identifies uniquely each node (Slave) in the network. A network can have up to 127 nodes (node ID 0 to 126). The 128th node is the virtual, broadcast, node (node ID 127). It is used to access all nodes simultaneously. Each node has two node addresses. The first node address enables access to the memory and memory mapped I/O register set. The second node address enables access to the system register set: command, flags, DMA counter and SPI.

The RX frame (Table 2 consists of 16 bits: start bit (always ’0’), interrupt bit INT, type bits TYPE[1:0], data bits DATA[7:0] and CRC bits CRC[3:0]. INT bit is set if one or more Slaves through which RX frame passed (including the Slave that generated the RX frame) has pending interrupts. DATA[7:0] hold valid data for response on ”Data register read” and ”Flags/SPI register read” commands. DATA[7:0] hold node ID and DATA[0] holds interrupt status (set if the Slave has pending interrupts) for response to all other commands. CRC is calculated over TYPE[1:0] and DATA[7:0] using the $x^4 + x + 1$ polynomial.

0	INT	TYPE[1:0]	DATA[7:0]	CRC[3:0]

Table 2. RX frame format

3.2 TpWIRE Scalability

The performance of the TpWIRE bus can be scaled with respect to the application requirements by increasing the number of lines from the 1-wire to a n-wire architecture. These enhanced architectures could be used in two different ways:

- One line is used to communicate with the Master, while the other lines are used to parallel transmit data;

- Each line is used to implement one 1-wire bus, thus having n parallel 1-wire transmissions.

The performance of these improved buses must be estimated in relation to the network traffic produced by a tuplespace middleware. To this purpose, a rapid prototyping methodology is required to perform this analysis avoiding useless time-consuming implementations.

4 Rapid Prototyping Methodology

This section describes the rapid prototyping methodology used for the evaluation of bus performance.

4.1 Tuplespace Infrastructure

Our implementation of the tuplespace middleware is based on the JavaSpaces technology [2]. As for other space-based technologies, JavaSpaces provides a mechanism for sharing objects between Java-based network resources, and it works as a virtual space (memory) accessible to both providers and requesters of remote objects. This allows the components of a distributed application to send and receive messages in the form of objects through the space. Each space takes care of all the details of the communication, such as synchronization or persistence.

According to the JavaSpaces terminology, a JavaSpaces server holds *entries*. Technically, an entry is a typed group of objects, expressed as a class that implements the `Entry` interface. The distributed programming environment is composed of one or more clients that communicate with the JavaSpaces server through the Java Remote Method Interface (RMI). Figure 3 shows the conceptual interaction between a Java client and a JavaSpaces server.

We have realized an implementation of our target system in Java for its rapid prototyping. The name of the space server class is `SpaceServer`. The Java prototype has been implemented naturally without using existing Java services: in this realization the Space Server communicates with the clients through Java RMI.

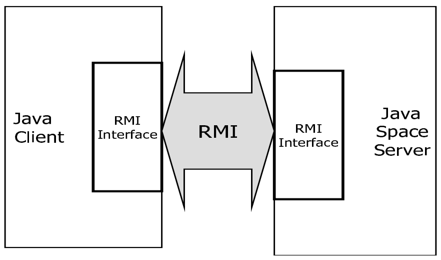

Figure 3. JavaSpaces Client-Server Interaction.

4.2 Tuplespace in a Hardware Context

The basic Javaspaces semantics does also apply to our context, where some of the participants of the distributed applications may be hosted by hardware devices. In particular, our objective is to access a space server from client applications hosted on Theseus boards.

To achieve this, it is necessary in our case to translate the prototype Java client into C++, to allow its execution on the Theseus boards, which do not support a Java Virtual Machine. This issue poses the problem of how to interface the Java Space Server with a C++ application. This is solved as shown in Figure 4. A Java/socket wrapper is introduced on the server side. Unix sockets are thus used to connect the board with the host running the space server.

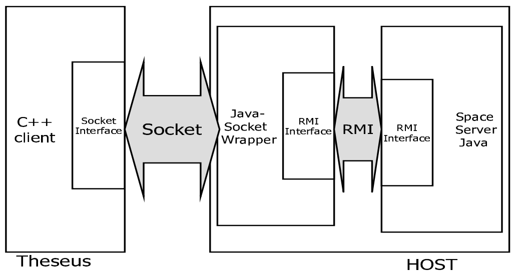

Figure 4. Board-Space Server Interface.

Using sockets, communication between the client and the `SpaceServer` relies on TCP-IP for information exchange and in particular, XML is used to represent data entries. RMI is still used inside the server, this time to interface the server with the Java/socket wrapper.

4.3 Network Infrastructure Integration

The use of sockets to connect a client to the space server implies the use of TCP-IP as a protocol, and then the use of the Ethernet as physical medium. This configuration has several advantages, mainly because of its natural software abstraction (i.e., UNIX sockets).

When connecting several clients with a server, however, we have to consider, as in our target application, that each client represents a board. In these cases, the connection of several boards with the server through a TCP-like network may not the best choice, for several reasons. First, the cost of such a connection may be too high; in fact, it would require the presence of active device (e.g., switches) which may not be amortized in some low-cost applications. Second, in some industrial applications (e.g., where some of the devices mounting the boards are moving or hardly accessible), it may not be feasible to setup a complete network infrastructure.

For these reasons, the Theseus boards used in our implementation support a more efficient type of connection which is more suitable for low-bandwidth and low-cost applications than TCP-IP. This connection is realized through a fully programmable multimode bus *TpWIRE* described in the previous section. The boards come with the TpWIRE default configuration consisting of a 1-wire serial bus that allows to daisy chain other boards.

Unlike TCP-IP, however, that has a natural simulation support through the sockets API, the simulation of a TpWIRE connection requires the development of this specific simulation support. Rather than writing a custom simulation engine, we have used the Network Simulator 2 (NS-2) [9] framework to simulate the TpWIRE connection between a client and the space server.

NS-2 is a discrete event simulator that provides support for the simulation of an IP-based protocol stack, in which a number of network and data-link protocols are available. NS-2 is part of the Virtual Inter Network Testbed (VINT) collaborative research project aiming at generating a set of network simulation tools to be used especially in the design and deployment of new wide area Internet protocols.

The NS-2 architecture provides easy extension to the users. Network configurations are specified in NS-2 via *programmable composition*, where configurations are specified through a script. To this purpose, NS-2 presents a two-layer programming model. The simulator kernel is implemented in C++; conversely, simulations are defined, configured, and controlled by a NS-2 simulation program written in a TCL-like scripting language. Thanks to this approach it is possible to clearly separate the simulation runs from the actual simulator, its design, maintenance, extensions and debugging.

Since NS-2 does not provide TpWIRE support, we have implemented the TpWIRE protocol model into NS-2. Technically, it has implemented by defining a new *agent* object *TpWIRE Agent*; nodes on the bus are connected through a link, using the TpWIRE bandwith and th relative real-time specifications. Agents build TX and RX packets (see Section 3) and put them on the link. The master slave has also been implemented in TpWIRE agent.

Resorting to NS-2 rather than a custom C++ software model of TpWIRE is justified by the many features offered by NS-2, in particular, the possibility of generating various traffic workloads that can be used to separately validate the model before putting it on-field.

The use of NS-2 poses however the issue of the software interface between the C++ client on the board, and the space server. Figure 5 shows the resulting software architecture of the client-server connection.

The TpWIRE bus has been implemented using two SystemC [10] nodes, one for the client side and the other for the server side, that communicate with NS-2 using standard UNIX shared memory. The *SC1* SystemC process communicates with the C++ client, executed on the Theseus board. The communication is realized through an interface based on the remote debugging features of `gdb` [11]. The use of SystemC allows to realize and integrated cosimulation scheme, where the HW (written in SystemC) is seamlessly integrated with the instruction set simulator (`gdb`). Further technical details can be found in [12].

On the space server side, the *SC2* process communicates with the space server using UNIX sockets as described in Figure 4. Internally to the host running the space server, the communication between the server and the Java/socket wrapper is RMI-based as usual.

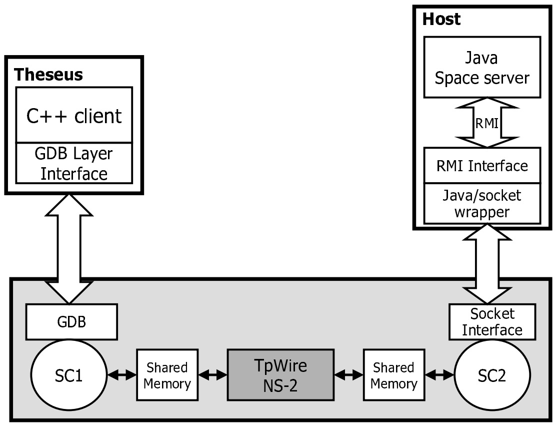

Figure 5. Board-Space Server Interface.

5 Bus Performance Estimation

In this section we estimate the impact of the tuplespace communication middleware at the “application” level, by analyzing the communication between two TpWIRE endpoints, with respect to some typical traffic profiles. The exploration of this “application” level is made possible by the the proposed prototyping methodology; by itself, NS-2 would allow to simulate the lower communication levels only.

Before measuring the tuplespace performance, we have to validate the NS-2-TpWIRE implementation. In order to compare with the real TpWIRE computation, an exact measurement is possible by using the NS-2 real-time scheduler. Under this operating mode, NS allows to tie events execution of the simulation kernel to real time. The test has been implemented by using the TpWIRE configuration, shown in Figure 6, consisting in a node, named *Slave1*, that communicates with another node, *Slave2*. We plugged a Constant Bit Rate (CBR) traffic generator on the *Slave1* node to send a 1 byte packet to the agent object that receives the data on the *Slave2* node. We then computed the time between the first transmission of data by the CBR and the time when the transmission is finished. The difference between the two times represents the exact number of clock cycles used by the TpWIRE protocol to transmit the data from *Slave1* to the *Slave2*.

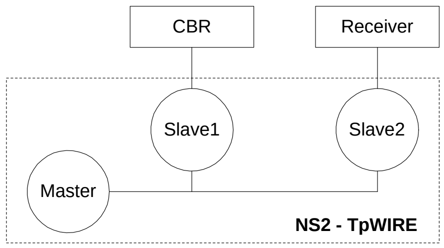

Figure 6. NS-2 scheme for TpWIRE model validation.

Using the same setup, we have calculated the real TpWIRE throughput, through the TpWIRE communication channel implemented in the Theseus system. Notice that this system ([13]) is currently provided only with the 1-wire implementation of TpWIRE .

TpWIRE	sec	Num. Frame
TpICU/SCM20	2.9	10000
NS-2	8.6	
TpICU/SCM20	28.9	100000
NS-2	86.0	
TpICU/SCM20	287.8	1000000
NS-2	860.0	

Table 3. Validation NS2-TpWIRE .

From these results we derived a “scaling factor” used to understand how close to reality is the NS-2-TpWIRE model for timing-accurate measurements to be used in our co-simulation environment.

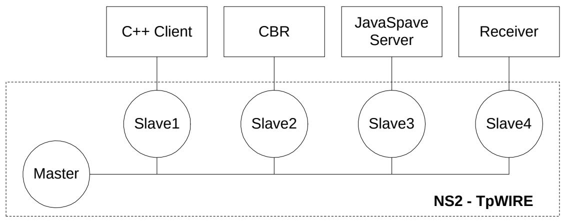

Figure 7. TpWIRE configuration.

After this first analysis we calculated the impact of tuplespace communication middleware on the TpWIRE bus, when data are exchanged through the NS-2-TpWIRE model. The case study, shown in Figure 7 includes a C++ client, attached on the *Slave1*, that communicates with the JavaSpace Server on the *Slave3* node, where a CBR attached on the *Slave2* node increases the traffic on the bus by communicating with a receiver modelled on the *Slave4* node. The traffic profiles applied to the tuplespace is represented by primitives for writing and removing *entries* from the space server. The C++ client executes a write-entry operation on the space; later on, a take operation is executed by the C++ client, which removes the entry just written from the space only if the entry lifetime is not out-of-date.

By increasing the traffic on the communication channel through the increase of the CBR value, the take operation does not positively result (returning the entry from the space), after a measured threshold of data traffic between the TpWIRE nodes (see Table 4). A potential 2-wire implementation of the TpWIRE can almost double the performance of the implemented 1-wire bus.

CBR	1-wire	2-wire
0 B/s	140s	116s
0.3 B/s	151s	122s
1 B/s	Out of Time	129s

Table 4. Estimation of the impact of tuplespace communication middleware on TpWIRE . Lease Time = 160s

In conclusion, the proposed rapid prototyping methodology, based on a combination of Java/C++/NS-2, allows to qualitatively and quantitatively estimate the performance of a low cost embedded bus, such as TpWIRE bus, in a complex tuplespace middleware. This estimation gave enough information to plan the complete development of the bus and the tuplespace.

References

[1] M. Munson, T. Hides, T. Fisher, K.H. Lee, T. Lehman and B. Zhao. “Flexible Internetworking of Devices and Controls” *IECON’99: IEEE Annual Conference of the IEEE Industrial Electronics Society*, San Jose, CA, December 1999,

[2] *Sun Microsystems*, “JavaSpaces”. http://java.sum.com/products/javaspaces

[3] *IBM*, “TSpaces”. http://www.alphaworks.ibm.com/tech/tspaces

[4] *Exor International*, “Theseus”, http://www.exorint.net/theseus/off/index.shtml

[5] D. Gelernter, “Generative Communication in Linda”. *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80–112, 1985.

[6] C. Carriero and D. Gelernter, “Linda in Context”. *Communications of the ACM*, Vol. 32, No. 4, 1989.

[7] *RogueWave*, “Ruples”, http://www.roguewave.com/developer/tac/ruple

[8] D.V. Moffat, “XML-Tuples and XML-Spaces”. http://uncled.oit.unc.edu/XML/XMLSpaces.html (1999).

[9] L. Breslau em et al. “Advances in Network Simulation”, *IEEE Computer*, pp. 59–67, May 2000.

[10] Synopsys Inc., *SystemC Users Guide*, version 1.2, 2001.

[11] *GDB Remote Serial Protocol,* http://www.gnu.org/manual/gdb-4.17.

[12] L. Benini *et al.*, “SystemC Co-Simulation of Multi-Processor Systems-on-Chip” *ICCD’02: IEEE International Conference on Computer Design,*, September 2002.

[13] *Exor International*, “TpICU/SCM20 - Theseus Program Integrated Control Unit / SCM20 profile”, www.embedding.net/icu