A connector-based approach for controlled data distribution in RTP architecture

Daniela Micucci - Francesco Tisato
D.I.S.Co. - Università degli Studi di Milano-Bicocca
Via Bicocca degli Arcimboldi 8, 20126 Milano, Italy
(daniela.micucci,francesco.tisato)@disco.unimib.it

Andrea Trentini
D.I.Co. - Università degli Studi di Milano
Via Comelico 39, 20135 Milano, Italy
trentini@dico.unimi.it

Abstract

As the size and the complexity of software systems increases, a proper decomposition approach is strongly required, in particular to address issues about “when” and “how” distribution and elaboration have to be performed.

In this paper we present RTP (Real Time Performers), a general-purpose architecture for the development of modular and configurable real-time distributed systems. It is based on basic and formally-modelled notions: projectors (data transferring components), state machines, command traces, topologies and strategists. The RTP decomposition approach is based on the component-connector pattern, refined with the specialization of the connector role. RTP rationale is the “externalization” and formalization of communication issues between components, with a strong separation between alignment mechanisms and policies.

1. Introduction

Nowadays almost every software system is built by composition of heterogeneous elements. Computational elements (usually called components) are connected one another by complex interactions. In the past, research work on component-based architectures was mainly focused on the internal structure and behaviour of components and on the communication interfaces between them. Terms like COTS (Commercial Off The Shelf), middleware, COM (Component Object Model), and CORBA (Common Object Request Broker Architecture) where defined to describe various composition models and architectures.

A research path, informally called the “vertical” one, follows the stratification aspect of architectures. In this field, works such as [16] and [18] describe very clearly the notions of conceptual, code, execution, concrete, and implementation architectures: terms defined to identify all the different levels of any general software architecture. We call it a “vertical” approach because it is usually graphically depicted in a top-to-bottom disposition. But another path is very (maybe more) interesting: the “horizontal” one. This is the field of researchers studying the compositional aspects of architectures. Indeed, composition models and techniques are fundamental for software reuse and maintenance at each meta-level [12]. Moreover, an important and useful notion in software architecture is the concept of “architectural style” [17] and [13]: a sort of higher-level set of patterns to categorize (and to reason about) different architectures. Styles are used to describe the compositional patterns (the overall organization) exploited to model architectures. In [17] the main focus is still aimed at components, but soon after, the same author, in the following [4] raises attention on the “connecting components”, the “glue”, the “in between” matter: the so-called connectors. In [4] the author admits that making a connector a first-class entity is not a new approach, but he also states that previous works offered a “fixed (albeit flexible)” set of primitive-connectors, thus giving birth to the notion of easily definable architectural connectors. Then, [1] focuses on the interaction between components by defining architectural connectors as explicit semantic entities. Connectors are specified by sets of protocols defining both the components roles and roles interactions. Protocols are described using CSP (Communicating Sequential Processes) language.

An important work concerning connectors is [6]. This work emphasizes the low level of understanding about both the interaction between components and how they may be composed into more complex interactions.

It seems that none of the aforementioned approaches takes into account, at the architectural level, the control aspect of data transmission.

The RTP (Real Time Performers) architecture presented in this paper, while still based on the component-connector pattern, tries to raise at the architectural level both the data and the data-distribution policies. The architecture is based on the following key considerations:

- a system is made up by computational components (Performers);
- computational components exchange information via
alignment components (Projectors);

• both computational and alignment components are
activated and controlled by supervising components (Strategists).

The identification of these well-distinguished components emphasizes the “separation of concerns” between information processing, information alignment, and their activation. This maximizes code reuse since computational and alignment components may be used under different policies and in different topologies. RTP is not only an architecture: its foundations have been turned into a framework supporting the implementation of domain related systems.

In Section 2 we will present the main features of RTP, using code or pseudo code from the framework when introducing architectural concepts; Section 3 deals with time issues; finally, Section 4 proposes an example of RTP exploitation.

2. The RTP Architecture

RTP is a general-purpose architecture for the development of strongly modular and configurable distributed systems. The architecture reifies the definition of a formal model (abstract machine) supporting the operational semantics of complex systems. RTP provides also a framework implementing the architecture and supporting the implementation of domain related systems. The framework has been developed using the Java programming language. Most of the underlying ideas of the architecture borrow from previous research activities and projects, in particular Kaleidoscope [15], RAID (Rilevamento dati Ambiente con Interfaccia DECT) [5], [7], and [8], RTO (Real-Time Objects), Architectural Reflection [19], HyperReal [11], and [2], and MAIN-E (MANufacturing & logistics INtegrated planning and control in SME multi-enterprise Environment). Specifically, Kaleidoscope architecture and its exploitation in RAID project (an indoor wireless environmental monitoring system with heterogeneous acquisition devices and applications) have contributed significantly to the definition of RTP architecture.

2.1. Computational components: Performers

A Performer is characterised by the following features:

• it has a local state;

• it can be modelled as a state machine where transitions correspond to the execution (atomic) of local transition functions (i.e., in O-O parlance, methods), whose definitions do not include the concept of time;

• it is passive, i.e., a transition is triggered by an external command.

The Performer environment is made up of variables belonging to two different sets: one containing local variables, i.e., only visible inside the performer; the other containing visible variables, i.e., the variables the Performer exports. A Performer has visibility over its environment only. Moreover, it has limited visibility over the set of its visible variables that has, in turn, two subsets: exported visible (writeable from the inside and readable from the outside) and imported visible (vice versa).

The proposed model is strictly local, i.e., Performers have a local name space and do not share state components (i.e., visible variables). There are several reasons for this choice: first, it encourages design modularity; second, it helps separating local behaviour, topology and strategy; third, it properly models the concrete architecture of a loosely connected distributed system.

In terms of architecture we defined a generic Visible interface representing instances from set of visible variables both exported and imported by a Performer. As shown in figure 1, this interface exports a get() method returning a value of type Object, and a set(Object) method that requires a value of type Object. Concrete visible variables have to implement the Visible interface. Visible implementations should model the data a Performer (possibly) needs to execute its task and the data it (possibly) produces for other Performers. Performer class (see figure 2) models a Performer. It is a generic command acceptor with unique name (it is a Nameable<

The accept(Command) method is invoked to trigger activities. Command represents a generic command to be executed by a Performer instance; the semantics of a Command instance is local to the receiving Performer.

Figure 1. The Visible interface definition

```
/** A visible exports get and set method that should be used
  * according to a “protocol” (e.g., for a “in” value only
  * the performer can call the setValue, ... */
  * public interface Visible {
  */
  /** Returns the value for this visible */
  * public Object getValue();
  /** Sets the value for this visible */
  * public void setValue(Object v);
  */
```

1 We do not deal here with shared memory multiprocessors. However, even in this case the existence of a shared memory implies synchronization problems. A shared memory is more properly modelled by a Performer and by suitable strategies. Thinking in terms of hardware, a shared memory is a physical component with a set of access “methods” and a set of suitable synchronization mechanisms. Our conjecture is that sharing does not physically exist

2 Every important entity in the system must be uniquely identified. In the framework we have the Nameable interface to tag these kinds of entities and a Naming System to track all these names.
2.2. Alignment components: Projectors

Projectors align information between Performers, i.e., they are responsible for data distribution. They transfer data from visible exported variables of a Performer into imported visible variables of another Performer. In other terms, a Performer can observe the exported visible variables of another Performer only if they are projected into its own imported visible variables. This does imply neither any assumption about why and when observation is done, nor about observation mechanisms (polling, asynchronous messages, and so on) whose definition is a matter of strategy only. Moreover, Performers need not be aware of Projectors and other Performers. Performers and Projectors define the system topology.

A Projector is a special kind (subclass) of Performer. Since the Projector role is to align data, its accept() method admits only a "synch" command (see figure 3). All the synch activities should be defined in the synch() method. Figure 3 shows also ConcreteProjector class: a simple Projector implementation that emphasizes the synch activity performed on visibles. More complex situations can be devised, e.g., either imported or exported visible variables may be queues, buffers.

2.3. System dynamics: Strategists

Till now nothing has been said about the overall dynamics of the system. We remark that a Performer is a passive component defining a local behaviour (i.e., what it must do on behalf of a command according to its local state), but it is not aware about when commands are generated. Accordingly, a Performer defines an alignment mechanism consistent with the system topology, but, in turn, is not aware about when the alignment must be performed.

Issues about when elaboration and information alignment have to be done are encapsulated inside Traces. Every Trace defines a part of the system behaviour in terms of partially ordered set of Requests. In turn, a Request is a pair (command, recipient) where recipient denotes either a Performer or a Projector. All the Traces described for a system fully define its overall behaviour. Our model allows a fully deterministic behaviour to be defined, if required by the application domain (vice versa, non-determinism simplifies the specifications by avoiding unnecessary constraints).

In terms of architecture, the Request class represents a request to be delivered. It contains a Command (the command to be executed) and a Name (the unique name of the Performer/Projector that should execute the request). The Trace class contains (see figure 4) an aggregation of Requests divided into two subsets: a sequence of executed Requests (immutable), and a set of Requests to be executed (mutable). The Trace next() method returns the next "performable" request, then moves it in the past trace. The Engine class is responsible for executing a Trace and dispatching Requests. It calls the Trace next() method, and delivers the Command to the proper Performer. The Engine is not aware of the system behaviour and it is paced by a general event generator called Ticker. It is remarkable to say that the Engine class will be reused (as is) even when introducing timings.

Finally, with the term strategy we define the set of future Requests as a function of the current trace and of the observable state of the Performers. The model accommodates both

---

3This is true under the condition of having non-deterministic guards.
on-line and off-line strategies. Furthermore, the model assists different strategies without affecting both topology and Performers definitions. For instance, an alignment can be triggered according to three different strategies:

- **push**: a Request is issued to a Projector when its source variable changes;
- **pull**: a Request is issued to a Projector on a change of a suitable exported visible variable of the Performer the target belongs to;
- **timed**: a Request is issued to a Projector according to specific timings.

Similar remarks apply to strategies generating commands for Performers.

A Strategist is a component implementing a strategy, i.e., it (possibly) observes the Performers state and generates the future Trace. It encapsulates any domain-related issues: it is the only entity that must be re-designed (better say, developed) when the application domain changes. The Strategist is a component implementing a strategy, i.e., it (possibly) observes the Performers state and generates the future Trace. It encapsulates any domain-related issues: it is the only entity that must be re-designed (better say, developed) when the application domain changes. The Strategist may modify a Trace upon system observation. Mechanisms and policies used to observe the system must be confined into the Strategist itself: it may poll interesting visibles or it may ask visibles to generate notifying events, if they are able to.

You can devise a global Strategist controlling the overall behaviour. At the extreme, there might be a Strategist for each Performer (i.e., each Performer has an autonomous thread).

### 3. Timing Issues

In RTP we reason in terms of discrete time, i.e., time instants are modelled by nonnegative integers. A **timeline** is a monotonic sequence of integers, meanwhile current time represents a point in a timeline. Current time is bound to change monotonically.

When introducing timing issues, Trace and Request have to be modelled consequently, i.e., they must be fleshed out with time information. A **Timed Trace** is a set of **Timed Requests**, i.e., pairs (request, interval) where interval is a segment of the timeline. Note that associating requests with intervals implies a partial ordering of the requests, i.e., on the Trace. Timed Requests are only eligible for being extracted from a Timed Trace when the current time falls inside the associated interval. A Timed Trace keeps track of the current time using **Virtual Clocks**. A Virtual Clock is an active component that is in charge of advancing the current time of a timeline. A tick is an increment of the current time of a timeline. A Virtual clock is associated to one **reference clock** (which, in turn, is a Virtual Clock). The **period** of a Virtual Clock is the number of ticks of the reference clock for each tick of the Virtual Clock. Informally, the period of a Virtual Clock defines the timeline advancement speed. In general, several Virtual Clocks may have the same reference clock. Therefore Virtual Clocks are logically arranged in a forest. When we need a system-wide clock the forest collapses to a tree whose root is the system clock. It is a clock whose advancement is driven by events that do not fall inside the model (for instance, physical interupt or keystrokes or simulation events).

Timed Traces rely on Virtual Clocks; therefore they are not aware of any kind of “absolute” time. The “absolute” execution speed of the Timed Trace defining the behaviour of a sub-system can be controlled from the outside. Of course, this holds for the future portion of the Timed Trace. The past portion of the Timed Trace, including the past periods of the Virtual Clocks, cannot be changed. A straightforward architectural mechanism is to define the past portion of a Timed Trace according to a Virtual Clock whose period is immutable. In many cases, the system clock has an immutable period - or, if you prefer, a period that can be assumed as immutable if compared with the “real” time.

The concept of interval deserves some discussion. Note that the amplitude of an interval in a past trace depends on the precision of the observation, whereas the amplitude of an interval in the future trace depends on the precision requirements of the application domain. In particular, future intervals whose upper bound is \( \infty \) can be exploited to define ASAP (As Soon As Possible) strategies in absence of Hard Real-Time constraints.

There may be several Virtual Clocks associated to different timelines. For simplicity, the period is assumed to be an integer \(\geq 1\). This implies that the “rate” of a Virtual Clock can change by discrete steps, and that the “granularity” of the reference clock, i.e., the duration of its tick, is not bigger than the granularity of the referencing Virtual Clock. These assumptions should not affect the discussion.
In terms of architecture, TimeScale class represents a timeline. It is characterised by a starting point, an end point, and a step (quantum increment). Meanwhile, Time class models current time. These are the basis classes for VirtualClock class definition. As VirtualClock class models a Virtual Clock, it updates a Time instance according to a TimeScale. It is characterized by a period: the number of reference clock ticks between updates of the Virtual Clock current time. Virtual Clock is a “time-event” generator for entities interested in current time. For this reason it is a specialisation of the Ticker class.

A Timed Request is defined as a subclass of the Request class, as shown in figure 5 (a UML [3] class diagram). TimedRequest is in association with one instance of TimeCoord class describing the validity interval for the Request. A TimedTrace is a subclass of the Trace class. It contains TimedRequests, and overrides the next() method with a time-aware implementation. A TimedTrace is aware of the current time by means of the associated VirtualClock. The Engine class is still the same, i.e. there is no need for a new entity.

4. An example

In this section we describe a simple example to show how RTP may be used. The context is taken from a real world pollutant monitoring system (RAID [5], [7], and [8]), modelled using RTP. Imagine a context with these Performers (as shown in figure 6):

- **Central System**, the “presentation” component, accepting the (PRESENT) command, and with only one $V_{in}$ imported Visible (exported Visible instances are not relevant for this example);
- **Monitor**, the computation component (it calculates the pollution concentration by reading 18 raw input values), accepting the (COMPUTE) command, with one $V_{out}$ exported Visible, and 18 $V_{in}$ imported Visible instances;
- $F_{s1}$ through $F_{s18}$, the field sensors, measuring environment raw physical values, and accepting (SAMPLE) command. They have one $V_{out}$ exported Visible.

The Performers are connected by the following Projector instances:

- **Pc**, connecting Monitor.$V_{out}$ with CentralSystem.$V_{in}$
- **$P_j$** (with $j = 1,...,18$), connecting $F_{sj}.V_{out}$ with Monitor.$V_{inj}$ Every Projector accepts only the (SYNCH) command.

We have instances of: MainSystem (the bootstrap code), Topologist (creator of the components and connectors), Strategist (creator and modifier of the trace), Engine (executing the trace), VirtualClock (keeping the pace), Central (the presenter component), Monitor (the pollutant calculator), FieldSensors (monitoring the environment) and Projectors (connecting the components). The bootstrap code is:

```java
Topologist.createTopology();
Strategist.createStrategy();
Engine eng = new Engine(tr1);
eng.activate();
vc1.activate();
```

Engine extracts Requests from the Trace and dispatches them to the right Performer. The Strategist places trace
subsequences triggering complete rolls of data from the FieldSensors up to the Central. Data is converted along the way by the Monitor component (examining every RawValue from \(V_{in}\) input and calculating PollutantValue to set in \(V_{out}\)). The Monitor component may also raise an alarm condition by setting a value for its Alarm. This Alarm visible is observed by the Strategist: when an alarm condition is raised, the Strategist can activate special behaviour. System behaviour is modified by changing either the VirtualClock pace and/or modifying the Trace content:

```java
//begin=0,end=10000,step=5,delay=5ms
createSubTrace(105); // utility method
// when in need of changing pace
vc1.setDelay(20); // to slow-down
vc1.setDelay(5); // to speed-up
// when in need of changing trace // (e.g.,
// adding a new sequence

createSubTrace(4000 instants after ‘‘now’’);
vc.getDelay();
```

5. Conclusions and future work

We have presented RTP, a architecture for the design of modular and complex systems. Our architecture is described in terms of components (Performers) and connectors (Projectors). Our Projector is very similar to the pipe in the [17] “pipe and filter” architectural style: we extended the pipe concept with a controlled pipe. The extension approach was taken also by [14] for the same goal (real-time data feeds), while we used a different technique: instead of relying on underlying features of a targeted operating system our approach is applicable to more general complex systems.

The RTP architecture enforces the components reuse since it operates a separation of concerns between computational (Performers), alignment (Projectors), and supervising (Strategists) components. Performers and Projectors are not aware of the actual surrounding environment and its dynamics. Moreover, RTP allows (behaviour and communication) strategy changes in the system without affecting components implementation. Strategies can be planned at the application domain level instead of relying on component-embedded strategies (that may only provide best-effort solutions since a single component does not know the overall system structure).

RTP is not only a mere model, it has been prototypically implemented in a Java framework. To further test the validity of these ideas, this framework is being exploited in two domain-specific systems: a vision/robotics 3D reconstruction application based on trinocular stereo vision; a complex distributed movement tracking system webcam based.

References