Eva: an Event-Based Framework for Developing Specialised Communication Protocols*

Francisco Brasileiro
Federal University of Paraíba
Computing and Systems Department
Campina Grande, Paraíba, 58.109-970
Brazil
fubica@dsc.ufpb.br

Fabíola Greve, Michel Hurfin,
Jean-Pierre Le Narzul, Frédéric Tronel
IRISA - INRIA
35042 Rennes Cedex
France
{fgreve|hurfin|jlenarzu|ftronel}@irisa.fr

Abstract

This paper presents a framework for the development of higher level communication protocols that provides extra functionalities (e.g. one-to-many ordered delivery, atomic delivery, etc.) not supplied by standard off-the-shelf lower level communication protocols (e.g. the TCP/IP suite of protocols). The framework is based on the event channel abstraction which allows circumventing the main drawbacks of the layered-based approach traditionally used to develop such protocols, whilst at the same time providing a flexible, simple and well structured way to implement them. The event channel service provided by EVA *establishes how entities that share the same address space interact. Then, the application designer has the opportunity to define the most appropriate lower level communication protocols that control the way entities that execute within different processes will interact. The framework specifies a way to accommodate these protocols and provides several standard protocol implementations. Further, it is described a development methodology for constructing applications (specialised communication protocols) on top of the framework. In designing the framework, we have followed the approach of using, whenever possible, well established concepts (e.g. event notification service, design patterns, etc.), thus the paper also discusses the utilisation of such concepts in improving both the efficiency and the structuring of the framework and of the applications to be built on top of it.*

1 Introduction

A distributed application can be generically regarded as being composed of a set of processes, normally executing at different hosts, that work cooperatively, and use communication protocols to exchange information. Different applications require different communication services that can range from standard unreliable point-to-point services (available in most off-the-shelf platforms) to specialised services provided by sophisticated fault-tolerant one-to-many communication protocols.

The work presented in this paper has been initially motivated by our need to build a particular type of specialised communication service, namely a *group communication* service [20], based on a generic agreement framework [8, 12]. To achieve this goal, we have defined an open framework (called EVA) which is suitable for designing and implementing any specialized communication service (consensus service, atomic broadcast, group membership service, etc.).

A number of systems that provide group communication services have been designed and implemented [7, 11, 16, 18, 26]. Although they present significant functional differences[1], they have all followed a layering strategy in their implementation. The goal of this paper is to propose an alternative approach to the design and implementation of specialised communication services in general, including group communication services.

Specialised communication protocols can often be seen as extensions of simple point-to-point communication protocols. As a layer-based architecture is frequently used to specify the basic communication protocols[2], it could seem natural to keep on designing higher level protocols in the same way (*i.e.* by adding new layers of protocols). To add a new functionality it suffices to create one or more protocol layers, each layer using the services of the adjacent lower

*This work is partially supported by a France Telecom grant 98 1B 123.

[1] It is out of the scope of this paper to present a detailed analysis on how these systems differ between themselves and in respect of the group communication service we have implemented.

[2] For example, such an approach has been adopted to specify the OSI and the TCP/IP architectures.

layer to provide an extended service to the adjacent upper layer, up to the last layer which provides the new functionality required. However, if on one side the structure imposed by the layer-based approach defines a way of developing a protocol that is clear and easy to understand, on the other side the need to accommodate the functionality of the protocol into layers that can only communicate with their adjacent layers imposes a very restrictive design model. Also, rather than being specified as a collection of well defined functionality layers, higher level communication protocols are normally presented as a description of the cooperating entities that together implement the functionality of the protocol, and the interactions between them [3, 4, 13]. Furthermore, layer-based architectures normally carry a performance penalty, as the information that concerns a particular layer has to be dealt with by each of the lower layers at both sender and recipient sides [15].

Group communication services whose implementation followed a layer-based approach have used advanced memory management techniques to reduce communication latency. In some systems, ad hoc optimizations (as for example, short cuts between non-consecutive layers) have also been used to further restrict the cost of layering [15]. Nevertheless, even with such improvements, a stack of protocols is still not flexible enough. In the development of specialised communication protocols, flexibility is a key issue when trying to match the specific needs of each protocol and improve its efficiency [15].

To avoid the drawbacks of a layer-based architecture whilst keeping the advantages associated with decomposing the problem into well defined and independent basic services, we propose an object-oriented architecture based on the event channel abstraction [19]. In the proposed architecture, the functionality of an application (namely, a specialised communication protocol) is implemented by a number of cooperating objects. Communication between these objects is achieved via the production of events (output data) by supplier objects, and the consumption of these events (input data) by consumer objects. A supplier object uses the service of an event channel to route the events it produces to any consumer object that has registered with the event channel its interest in consuming that particular type of event. The event channel decouples suppliers from consumers yielding the sought flexibility.

The event concept is well suited to model messages that contain data fields. Also, activities performed by the objects that implement the application can be triggered by the consumption of events. Indeed, as we show in this paper, an event-based architecture is a simple and elegant way to fill in the gap between the specification of the application and its implementation by a set of cooperative distributed objects that interact via the production and consumption of events. This approach allows the construction of highly modular and reconfigurable applications whose components can be executed in parallel.

The remaining of this paper is structured as follows. The event-based architecture discussed above has been materialised by a framework, whose main constituents are presented in Section 2. Section 3 describes a methodology for the utilisation of the framework in the development of applications. Then, in Section 4 we discuss issues related to the efficiency of the framework and the applications to be built on top of it. Section 5 gives an overview of EDEN, a group communication service based on the EVA framework. Section 6 contrasts our efforts with related work. Finally, Section 7 brings our concluding remarks.

2 The Eva Framework

The EVA (EVent-based Architecture) framework is composed of a set of Java classes that provide the basic functionality of most objects a programmer will need in implementing an application. The application can then be built via a simple inheritance mechanism that allows the incorporation of application specific behaviour into extensions of the basic classes provided. In some cases the basic classes themselves can be directly used. In the following subsections we discuss the main functionalities provided by EVA.

2.1 The Event Communication Service

Applications built on top of EVA are composed of a set of cooperating objects, named *components*, that communicate with each other via the exchange of special objects named *events*. A particular component is itself formed by cooperating objects, named *entities*, which may also communicate via the exchange of events. A component contains entities that share a single address space. An entity may be a supplier as well as a consumer of events, and may be subordinated to, *i.e.* be part of, an unlimited number of components. Consumer entities must register with the appropriate components to which they have been subordinated, their interest in being notified of the production of a particular type of event. Each component is responsible for the management of an *event channel* that routes an event produced by one of its subordinated supplier entities to all of its subordinated consumer entities that have registered interest in receiving a notification for that type of event.

In summary, the event communication service is based on the following three main concepts: i) events; ii) entities; and iii) components. An event is a "container" object used to convey data from supplier to consumer entities, an entity is a "worker" object that interacts with other entities to implement part of the functionality of the application, whilst a component is both a "structuring" object that assembles related entities together, as well as a "manager" object that coordinates the interactions of its subordinated entities. A component is itself an entity that can be subordinated to other super-components.

2.1.1 Eva Events

An event is an object whose corresponding class must implement the *Event* interface. The type of an event is defined by another object which is associated with it. The *Event* interface defines the operation *getType* to access the type of an event at run-time.

The *EventImpl* class, which is the default implementation of an event provided by the framework, uses the intrinsic association existing between an object and its class to implement the association of an event to its type; in other words, an event which is an instance of class *EventImpl* is an event of type *EventImpl*. The application designer might want to implement this association in a different manner, to allow, for example, instances of the same class to be treated as events of different types.

The implementation of a new type of event required by a particular application is normally achieved by defining a class that inherits from the *EventImpl* class provided. The new class simply defines the extra data fields and operations that are specifically required by the application.

In addition to the *Event* interface, EVA defines interfaces for describing the events that will be consumed or supplied. A consumer entity uses an instance of a class that implements the *ConsumableEventDescriptor* interface to precisely describe how it will consume events of a given type. Similarly, a supplier entity uses an instance of a class that implements the *SupplyableEventDescriptor* interface to describe a particular type of event that it produces.

2.1.2 Eva Entities

Within the framework an entity is an instance of a class that inherits from the basic *Entity* class. Further, consumer entities must implement the *Consumer* interface, whilst supplier entities must implement the *Supplier* interface. The *Consumer* interface defines the operation *consumeEvent* which is the call-back function invoked by the event channel of a component for delivering an event to the target consumer entity; it takes as a parameter the event that has been produced by one of the supplier entities subordinated to the component. The *Supplier* interface, on the other hand, defines the operation *consumptionNotification*; this operation is introduced solely to improve the performance of the event channel service and is discussed in Section 4.

A number of implementations of entities are provided by the framework. Some entities implement only the *Consumer* interface, others implement only the *Supplier* interface, whilst others implement both interfaces. There are both passive and active entities. An independent thread is associated with each active entity. For instance, most of the consumer entities provided by the framework are active entities. Events consumed by these entities are buffered for future processing, *i.e.* the invocation of the *consumeEvent* operation on entities of these classes simply places the given event into an associated buffer; the code executed by their corresponding threads continuously verify the presence of a new event into their associated buffers, thus allowing events to be processed asynchronously. Some entities are also able to execute a periodic operation, *i.e.* it is possible to associate with them a time interval within which two consecutive invocations of a specified operation on a specified object should be executed. The main entities provided by the framework and their characteristics are presented in Table 1.

Many communication protocols have been specified by their authors within the context of a message passing model. As we show in Section 3, the implementation of such protocols using the EVA framework is rather intuitive. Basically, each type of message used in the original protocol is derived into a type of event. Also, since in these protocols, event production (*i.e.* the sending of a message) is triggered either by the consumption of another event (*i.e.* the reception of a message) or by the passage of time (*e.g.* the expiration of a time-out), most entities that implement the functionality of the protocol can be mapped into instances of the entity classes already provided by the framework, or yet into instances of user-defined classes that inherit from the aforementioned classes.

2.1.3 Eva Components

As mentioned before, cooperating entities require a common event channel service to allow the exchange of events between them. Instances of the *ComponentEntity* class are responsible for the provision of such service. Each component entity defines its own private event channel service. To use the service of a particular event channel, an entity must first subordinate itself to the corresponding component by invoking the operation *addEntity* of the component interface, giving as a parameter a reference to itself.

A component is also an entity that can consume and produce events. Registering a component with one of its super-components for consuming/producing a particular type of event is equivalent to registering all its internal entities that have themselves registered for the consumption/production of that type of event. Thus, in general a component can be treated as a black box whose interface is defined by the events it consumes and those it produces. Moreover, an upper level component is responsible for controlling the actual interface of its subordinated subcomponents; this is accom-

[3] A component entity is a container of other entities, thus it will possess a particular characteristic only when it contains at least one entity that possess this characteristic.

Class name	Characteristics			
	Consumer	Supplier	Active	Periodic
ConsumerEntity	✓		✓	
TimedConsumerEntity	✓		✓	✓
TimedSupplierEntity		✓	✓	✓
ConsumerSupplierEntity	✓	✓	✓	
TimedConsumerSupplierEntity	✓	✓	✓	✓
CouplerEntity	✓	✓		
TimedCouplerEntity	✓	✓		✓
ComponentEntity[3]	✓	✓	✓	✓

Table 1. EVA **entities and their associated characteristics**

plished by selectively registering the subcomponent for the consumption/production of any event that should be consumed/produced.

In summary, in addition to managing the interactions of its internal entities, components play an important role in structuring applications. They provide a way of connecting related entities, hiding them from unrelated entities, and yielding application design more modular. EVA also provides auxiliary classes whose instances can be used to build and configure components with different functionalities. These classes implement the *Builder* design pattern [6].

2.1.4 Inter-Process Communication

The event channel mechanism presented requires the communicating entities (suppliers and consumers) to be subordinated to the same component entity, thus it can only be used by entities that share the same address space. Most systems that implement the event channel abstraction use the same mechanism to allow both intra-process and inter-process communication (see [1], for example). This design choice normally forces the utilisation of predefined lower level communication and data representation protocols to transfer data across different address spaces, possibly residing in different hosts, with different architectures. In implementing higher level communication protocols, choosing the appropriate underlying communication service is commonly a critical issue [17]. Therefore, we have followed a strategy that offers such flexibility to application designers.

The EVA framework provides special types of consumers, named *network notifiers*, that are able to transmit special kinds of events, named *remote events*, from one process to another. Similarly, special types of suppliers, named *network listeners*, are also provided to receive remote events and produce them locally. Notifiers and listeners instantiated at different processes are therefore responsible for the implementation of the linkage between the components to which they are subordinated. A pair of associated notifier and listener entities can be seen as a single distributed entity that consumes an event produced at one component and produces it at a remote component.

Three basic flavours of notifiers and listeners are provided: *datagram unicast*, *datagram multicast* and *stream*. Datagram unicast notifiers together with their listener counterparts provide an unreliable point-to-point communication service. Similarly, datagram multicast notifiers and listeners provide an unreliable one-to-many communication service. Finally, stream notifiers and listeners provide a reliable, FIFO, point-to-point communication service. Flexibility is attained by allowing application designers to use the provided classes, extend them, or create entirely new notifier and listener classes that will precisely fulfil their needs.

Events that are defined by applications, and that will be transfered across processes, must inherit from a particular class named *RemoteEvent*. Basically, the difference between an event and a remote event is that the latter must define operations that are able to marshall an event before it is sent, and unmarshall the event after it is received. These operations allow the correct re-constitution of an event at a remote process.

2.2 Synchronous Interaction

The event channel communication service is particularly suitable to model the common one-way interactions between the entities that implement a communication protocol. Allowing cooperating entities to communicate via an event channel service provides a very flexible programming model, since entities are entirely decoupled and interaction can be made completely asynchronous, *i.e.* non-blocking. As we have shown, EVA provides straightforward facilities to implement applications in accordance with this model.

However, in some cases, it can also be desirable to allow synchronous (*i.e.* blocking) interactions between some entities within a particular application. Interactions modeled by a client-server communication pattern are a typical example of such situations. In an asynchronous programming model, this kind of interaction should be implemented in the following way: both server and client entities produce and consume events; the server consumes request events produced by the clients and produces the corresponding reply events; the clients, on the other hand, produce request events and consume their respective replies; further, at both client and server sides, internal synchronisation mechanisms between their consumer and supplier parts must be implemented.

The approach described above leads to a design that is not natural. In fact, when entities execute within the same address space, synchronous interactions are naturally implemented via a procedure call mechanism. However, allowing an entity to directly invoke an operation implemented by another entity creates a static coupling between them that may rule out the possibility of using them as black boxes that can be easily reused. EVA provides, therefore, a mechanism to decouple entities that need to interact synchronously, increasing design flexibility. Component entities provide an interface that allows their subordinated entities to register the operations that they want to make available to other entities that share the same component. Components also provide the appropriate interface to allow access to the operations registered. In EVA these operations are termed *services*.

To register a service, an entity must invoke the *registerService* operation on the appropriate component entity, giving as parameters an identifier for the service being registered and a reference to a *handler* object. A handler object encapsulates references to an object and to one of its operations (in this case, the object and operation that actually implement the service).

Within a given component a service has a unique identifier. Thus, an entity can invoke any service that has been registered with any of the components to which it is subordinated, by simply invoking the *invokeRequest* operation on the appropriate component, giving as parameters the identification of the service and, if required, the parameters for that particular invocation of the service. The *invokeRequest* operation returns an object that can be used to carry the result of the service invocation. Requests are implemented via synchronous calls to their corresponding operations. An invocation of a service that has not yet been registered raises an exception.

Components can also register the internal services they manage with a super-component to which they are subordinated. This allows entities that are external to the component, but that share with the component a common super-component, to access these services.

3 A Development Methodology

In this section we present a methodology for the development of applications using EVA. We start by describing a simple application and then we discuss its design in the context of the framework.

3.1 A Failure Detector Component

A number of communication protocols designed to be executed over an asynchronous run-time environment assume that each process that implements the protocol has access to the services provided by an failure detector [4]. Informally, such a failure detector allows a process to suspect (although sometimes wrongly) that another process has crashed. One conceivable implementation of such a service would require the association of a local failure detector module with each process; these modules would exchange "I am alive" messages and would use a time-out mechanism to trigger suspicions of (possibly) crashed processes.

In the following subsections we consider the design of a component entity that implements a failure detector module with the following characteristics: i) at any time, local entities can query their associated failure detector component to have access to the list of processes that are currently suspected of having crashed; ii) the failure detector issues an "I am alive" message approximately every T_e units of time; and iii) if the failure detector does not receive an "I am alive" message from the failure detector associated with a process within an interval of time of duration T_r, $T_r > T_e$, then it suspects that process of having crashed [21].

3.2 Component Design

The design of an application, or a component of an application, using EVA is performed in a four-step process. The first step is to identify the entities that will be required to solve the problem. Then, one must identify the events (mainly messages) that these entities produce and consume. The third step is concerned with the identification of any service that must be provided. It consists of defining which entity will be responsible for the provision of each one of the required services. Finally, the interactions between entities must be studied and a way to assemble them together into a number of subcomponents must be defined. Note that this step may include establishing how the entities executed within a process will interact with those executing within another process. This in turn requires identifying, as part of the first step previously described, the appropriate listener and notifier entities that will allow the cooperation of the different processes that implement the application or component.

Not by accident, the informal application description presented in the previous section makes the design task extremely simple. Indeed, as previously pointed out, the applications we are interested in implementing are often described in terms of their constituent entities and the messages (events in EVA terminology) they exchange. The next subsections follow the steps presented above to present a design for the application described in Subsection 3.1.

3.2.1 Identifying Entities

From the description of the failure detector given earlier we can identify two main entities that will implement the functionality of the failure detector at each process: one that consumes "I am alive" messages produced by the failure detector components executed by the other processes, and periodically checks if there are processes that have crashed; and another that periodically produces "I am alive" messages to be consumed by the other components. Note that the first entity only consumes events, the second entity only supplies events, and both entities perform time-triggered actions. These entities can be implemented by instances of the *TimedConsumerEntity* and the *TimedSupplierEntity* classes provided by the framework (see Table 1). It suffices to instantiate them with the appropriate handlers for their corresponding periodic functions and the time intervals T_r and T_e, respectively.

Since the above entities produce/consume events that are consumed/produced within another process, we must also have at least one listener entity and one notifier entity within each component. Notice that since we do not make the assumption that the failure detector is reliable, there is no need to guarantee that all events produced by one failure detector component will be received, and therefore consumed, by all other components that have not crashed. Therefore, the required functionality, namely an unreliable one-to-many connection between all processes executing a failure detector component, can be implemented by equipping each component with one instance of the *MulticastDatagramListener* class and one instance of the *MulticastDatagramNotifier* class, provided by the framework.

3.2.2 Identifying Events

Identifying the events is an even simpler task. In general, the application description presents all the messages that need to be exchanged by the entities that implement it. Then, it suffices to map each type of message exchanged into a type of event. In the example presented the only messages exchanged are "I am alive" messages, thus only one type of event needs to be defined.

Events that are to be transmitted are implemented by defining a new class that inherits from the *RemoteEvent* class. In this case, apart from any extra data fields and operations required, the derived class must also provide implementations for the *marshall* and *unmarshall* operations necessary to correctly transmit data across different execution platforms.

3.2.3 Defining Services

Some entities within an application might need to interact in a client-server way, thus the asynchronous interaction provided by an event channel is not the most appropriate way to model this type of interaction. As discussed before, within EVA these interactions are modeled as synchronous services.

In the description of the failure detector previously presented, it is easy to identify the single service required. The failure detector must provide a service that allows other entities to have access to the list of processes that are currently suspected of having crashed. Also, the entity that consumes "I am alive" events produced remotely can be clearly elected as the most adequate entity to implement this service, since it is the absence of remote "I am alive" events that triggers suspicions.

3.2.4 Identifying Interactions

Now that all entities, events and services have been identified, the final step in the design is to define how the entities are to be connected together. This is achieved by subordinating the defined entities to the appropriate component entities.

A straightforward way to connect the entities is to use a single component and subordinate all entities to this component. However, this simplistic solution might not be the best choice. For instance, considering our example, this approach would lead to the definition of an entity of the *ComponentEntity* class - let us name it *failureDetector*, and the following four entities that are subordinated to it: a *detector* (an instance of the *TimedConsumerEntity* class), an *emitter* (an instance of the *TimedSupplierEntity* class), a *listener* (an instance of the *MulticastDatagramListener* class), and a *notifier* (an instance of the *MulticastDatagramNotifier* class). Both *notifier* and *detector* would register with the *failureDetector* component to be notified of the production of "I am alive" events. Figure 1 pictures this structuring.

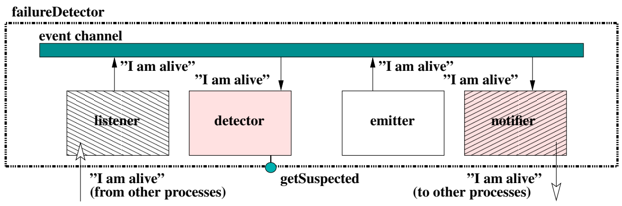


Figure 1. Entities structuring in a failure detector component

Note however that the *notifier* is only interested in those events supplied by the *emitter*, *i.e.* the "I am alive" events that should be transmitted to the components executing on the other processes; similarly, the *detector* is only interested in those events produced by the *listener*, since those correspond to remote “I am alive” events. Nevertheless, as all entities are subordinated to the same component (and therefore to the same event channel), both consumer entities will consume all events produced. Requiring consumers to deal with useless events, that are discarded after being consumed, increases application latency. However, this undesirable behaviour can be easily circumvented by isolating independent entities into distinct components. Figure 2 shows how the *failureDetector* could be structured to avoid the aforementioned inconvenience.

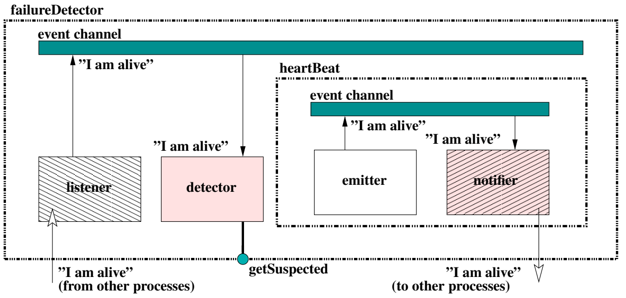
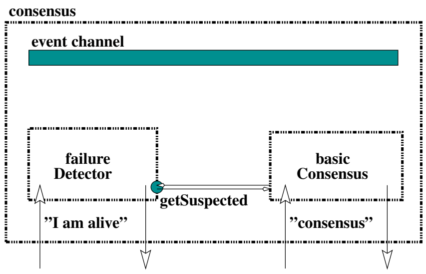


Figure 2. Isolating unrelated entities within a failure detector component

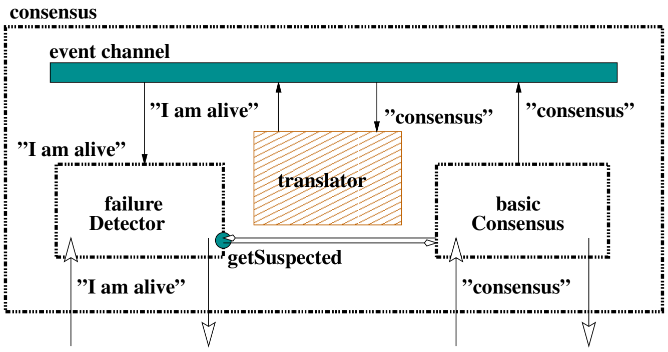
With the structuring depicted in Figure 2 the *heartBeat* component works completely independent from the other entities that implement the *failureDetector* component. In particular, provided that the *failureDetector* component does not register the *heartBeat* subcomponent as a supplier of “I am alive” events, events produced by the *emitter* of the *heartBeat* are not routed externally to the *detector* of the *failureDetector*. Similarly, provided that it does not register the *heartBeat* subcomponent as a consumer of “I am alive” events, events of this type produced by its *listener* are not routed internally to the *notifier* of the *heartBeat*. The *heartBeat* component can be seen as a black box that periodically produces “I am alive” events to be consumed by remote entities.

3.3 Achieving Flexibility via the Use of Components

As hinted before, EVA components allow very flexible design. We illustrate this via an example. Achieving consensus on a value is a fundamental problem in distributed applications (*e.g.* agreeing on the delivery order of a set of messages received), and many solutions proposed for this problem require the use of a failure detector [4, 12, 13]. Let us assume that we have a component named *basicConsensus*, that can be used to implement a consensus protocol provided that it has access to the service of a failure detector. An implementation of a solution for the consensus problem, would then consist of subordinating a *failureDetector* component and a *basicConsensus* component to a common super-component. Further, the *failureDetector* subcomponent must register its *getSuspected* service with the *consensus* super-component, so that this service can be used by the *basicConsensus* subcomponent (see Figure 3(a)).



(a) A simple consensus



(b) Improving the quality of service of the failure detector

Figure 3. Consensus components

Consensus protocols involve the exchange of messages among the processes that participate in the protocol. Events corresponding to these messages could also be used to inform the *failureDetector* components that a particular process has not crashed. This would improve the quality of service of the failure detector, without increasing the number of messages exchanged.

For the sake of simplicity, let us suppose that in the design of the *consensus* component, all messages that are exchanged have been mapped into a single type of event named a “consensus” event. Now, the problem we are faced is how to allow a *failureDetector* component to consume “consensus” events. The solution is the addition of an entity that translates "consensus" events into "I am alive" events. EVA provides passive entities, named *couplers*, that are tailored to perform this type of task. Figure 3(b) shows how this can be achieved.

As it can be seen, EVA components provide a very simple and flexible way to combine subcomponents together to implement a particular functionality. Components also offer an effective way to reuse both design and coding. In a typical layered-based architecture translators such as the one presented above can also be introduced. However this might imply in changing the behaviour of layers, which in turn may also force adjacent layers to change and make reuse more difficult.

4 Efficiency Issues

The performance of many distributed applications is considerably affected by the efficiency of the communication protocols they use. Therefore, minimising latency is normally a key requirement for most communication protocols. So far, we have discussed how EVA provides a flexible framework for the development of communications protocols. We have also taken special attention on building a framework that can also yield latency-efficient solutions. In the following subsections we analyse how potential sources of overheads have been limited or eliminated in the design and implementation of the framework.

4.1 Eliminating Potential Bottlenecks within Components

4.1.1 Buffering Consumers

An event channel can become a bottleneck in the application if the number of entities that use it to interact is considerably large. This problem can be circumvented by allowing supplier entities to dynamically maintain a table with the references to the consumer entities that have registered interest in the events the former produce [10]. With these references, suppliers can themselves directly deliver to the corresponding consumers the events they produce.

In this case, instead of using the information it maintains to route events produced to the consumers that have registered interest in consuming them, the event channel uses this information only to keep the appropriate suppliers aware of which consumers are interested in consuming the events they supply. Supplier entities must register with the event channel to specify the types of events they will produce. The *Supplier* interface implemented by supplier entities defines the *comsumptionNotification* operation, which takes as parameters a reference to a consumer object and an event type; this operation is used by an event channel to make a supplier entity aware of any consumer entity that has registered to consume a particular type of event that it produces.

4.1.2 Buffering Services

Components are also responsible for decoupling entities that interact synchronously. Thus, this can also be a contention point for applications. Further, since every synchronous interaction requires extra calls to the component object that manages a particular service, and to the operation that actually provides the service, if synchronous invocations to a particular service are frequent, the extra overhead incurred might be significant.

To solve this problem EVA allows a client object to have access to the handler object associated with a particular service. The *ComponentEntity* class provides a *getService* operation that takes a service identification as a parameter and returns a reference to the associated handler object. An entity that possesses such a reference can then invoke the *getServer* and *getOperation* operations on the handler object to obtain the information required to directly invoke the service.

4.2 Speeding Up Event Manipulation

4.2.1 Efficient Generation of Events

To produce an event a supplier must first create an object that is an instance of the class representing the event to be generated. Depending on the frequency with which this is done, instantiating new objects each time an event must be generated can consume a lot of time. Further, in many applications it is common to have a situation on which events are frequently generated for being used during a short time and then destroyed. The EVA framework provides facilities to speed up the creation of events within this context.

A special type of object, called a *Factory* is used to create events of a particular type. Factory objects efficiently manage the creation of events. The framework also defines a *Factorable* interface that must be implemented by all classes that want to allow their instances to be created by a factory. In particular, the *EventImpl* class implements the *Factorable* interface. Factorable objects created by factory objects possess a link to their corresponding factory that allows the application to explicitly return them to their original factory when they are no more needed. Returned objects can later be reused by any supplier with access to the same factory object.

In addition, EVA provides every supplier with a *FactoryPool* object. Factory pools are collections of factory objects that can be used to create different types of factorable objects. Whenever a new event is registered to be produced by a particular supplier, its associated factory pool is augmented with a factory that is able to create instances of the corresponding event.

4.2.2 Sharing Events

An event that is produced by a supplier entity might be consumed by several consumer entities. For instance, in the *consensus* component presented in Figure 3(b), a "consensus" event produced by any supplier entity subordinated to the *basicConsensus* component will be consumed by all internal consumers that might have registered with the *basicConsensus* component for consuming this type of event, as well as by the *translator* entity, and by the *failureDetector* component[4]. Further, a common behaviour of consumer entities is to use the data contained into the event consumed to either modify its local state, or to initialise the data fields of a new event that it will produce. In both cases the consumed event is not changed and can be returned to its corresponding factory soon after its consumption.

To provide better performance, the framework assumes that events are read-only objects that can be shared. The *Event* interface defines the *getShareableCopy* operation that allows a supplier entity to increase a reference counter associated with an event it produces. To return an event it has consumed, a consumer entity must invoke the *returnShareableCopy* operation of the *Event* interface. This operation decreases the reference counter of the event, and if no other entity is currently dealing with a copy of the event, *i.e.* if the reference counter is equal to zero, the event is returned to its factory for being reused. With this mechanism, copies of events are only performed when a consumer needs to change the data fields of an event that it has consumed (possibly to use the modified event as an event that it will later produce).

4.2.3 Efficient Transmission of Events

Marshalling and unmarshalling operations are normally costly, thus if an event that has already been marshalled and transmitted to another process needs to be transmitted again, it would be interesting to have a manner to avoid the unnecessary marshalling and unmarshalling of these events. The EVA framework allows this by defining the notion of events that can be associated with remote events.

A previously marshalled object can be stored into instances of a special class called *FragmentedRemoteEvent*. Thus, associated events are implemented as instances of this class. Remote events possess special data fields that can be used to store an unbounded number of such objects. Marshalling and unmarshalling operations are applied to none of the associated events of a remote event.

[4]More precisely, by the *detector* entity of the *failureDetector* component (see Figure 2).

4.2.4 Filtering Events

In Section 3 we have shown how judicious use of the component concept could help in avoiding an unwanted event from being routed to a particular consumer entity. However, for some cases, this approach is not flexible enough and cannot be used to attain a suitable solution.

EVA provides an explicit filtering mechanism to discard unwanted events. This is achieved by an object which is an instance of a class that extends the *Filter* abstract class provided. The extended class must define the *discardEvent* operation that takes an event as a parameter and returns a boolean value that will indicate whether that event should be discarded or delivered to the corresponding consumer.

The event descriptor object used by a particular consumer to register its interest in consuming a particular type of event may carry, if necessary, a reference to the filter object to be used over events of the type being registered. For each type of event to be consumed by a particular consumer, if a filter object has been defined, then its *discardEvent* operation is applied over the event, just before its delivery, allowing unwanted events to be discarded before their consumption.

In the improved *consensus* component shown in Figure 3(b), since both local and remote "consensus" events are going to be routed to the *translator* entity, the translated "I am alive" events will correspond to both local and remote events. Thus, the *detector* entity of the *failureDetector* component will be forced to deal with useless local "I am alive" events. A more efficient implementation should incorporate a filter object to discard local events consumed by the *translator*.

4.2.5 Defining Priorities for the Consumption of Events

The use of priorities is another way to speed up the manipulation of events. For instance, when an event carries information that may cause other events to be handled faster, or even to be discarded, associating a higher priority with this type of events can improve the performance of the application.

Some of the consumer entities provided by EVA allow the definition of different priority levels for different event types. Again, the priority for the consumption of a given type of event is defined at registration time, thus this information is also carried by the correspondent event descriptor object.

5 A Concrete Use of the EVA Framework

Within this section, we provide a short description of a group communication service called EDEN and developed at IRISA. More precisely, EDEN includes several high level protocols developed of course with the EVA framework . EDEN provides the main functionalities of a group communication service. More precisely, it implements both an atomic broadcast service [4] and a membership service [8] that can be used in correlation to provide a view synchrony semantics. Both services are considered as particular agreement problems and thus can be solved thanks to a common agreement mechanism [12].

The group paradigm is a natural solution to simplify replication management. Replication is a key mechanism to provide a higher degree of fault-tolerance in distributed systems. To tolerate crash failures, a critical server is replicated on different sites of the distributed system. If sites fail independently, an invocation to the fault tolerant server will succeed even if some replicas have crashed. In the active replication scheme, all copies of the replicated server concurrently execute the same task: each replica handles invocations and sends a response to the client. Thus the failure of a replica is not perceived by a client which waits until it receives the first response.

Group services allow a collection of related objects to be considered externally as a single logical entity. Each active replica is called a group member. An object group reference allows to refer to the whole group without being aware of its components. In addition to this "encapsulation" property, group services include different primitives used by group members to coordinate their activities. In particular, group services provide a fundamental communication abstraction called *atomic broadcast*: when an object invokes an operation using an object group reference, the request is in fact forwarded to all the members of the group and delivered either by all the non-crashed members or by none of them (atomicity property). Furthermore all the requests are delivered in the same order (total order property).

A group is not a static entity. Objects are dynamically inserted or removed from the group. The *join* operation is invoked by an object which wishes to become a group member. The *leave* operation is invoked by an object when it wants to leave the group. The local vision of the group composition at a given time is called a *view*. A *membership* service tracks changes in the group composition that result from explicit *join* and *leave* operations as well as implicit leaves due to the crashes of some replicas. Whenever a change occurs, each non-crashed member of the current group has to be aware of the new membership. Installation of a new view satisfies the *view synchrony* property which states conditions on both the installed views and the set of invocations that are delivered between two successive view changes.

Thanks to these group communication services and their associated properties, replicas can perceive the same sequence of requests in a concurrent manner while keeping their internal state fully consistent. Of course, this state consistency can only be ensured if replicas behave deterministically. Figure 4 provides a general overview of the group communication service. Only the main components are described.

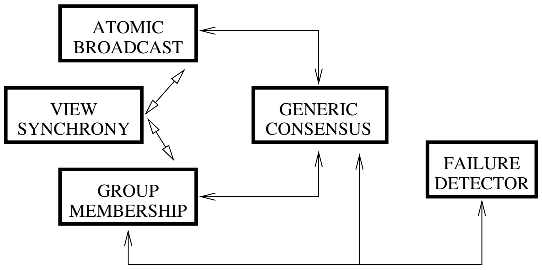


Figure 4. Overview of Eden

The development of all these protocols has been greatly simplified thanks to the EVA framework. During the development of this sizeable set of services, the intensive use of the different functionalities (events, factories, filters, queues, priorities, ...) has allowed to validate our proposal and to confirm that this framework offers sufficiently many simple mechanisms which can be operated in a natural way.

An event-based communication approach is well suited to develop a consensus-based group communication: many events are produced by one entity and consumed by several others. For example, a decision event is generated by the consensus entity and consumed asynchroniously by nearly all the other components (atomic broadcast, membership, view synchrony, ...). Moreover, such a strategy allows to dynamically integrate additional agreement services, as for example a leader election service. We are currently investigating the possibility to provide a set of services (EDEN) which can be automatically adapted to fit exactly the agreement requirements within a group of machines.

6 Related Works

In the past, several works have been done to simplify the development of distributed communication softwares. To avoid the drawbacks of using systematically static protocol stacks, solutions such as x-kernel [14], Coyotte [2], or ACE (Adaptive Communication Environment) [22] have proposed a new way to design network softwares. Like the EVA framework, these solutions aim to allow the development of modular distributed communication software which can be reused and configured to obtain the right combination of basic protocols that fits exactly the needs of the application. Whereas the conventional architectures use a small and fixed set of complex protocols architectured in a simple and static way (a stack), the above solutions suggest to use multiple simple protocols which are architectured in a complex and reconfigurable protocol graph. Additionally, these solutions also provide a support for protocol development.

If its goals are similar, the object oriented EVA framework differs from these previous solutions mainly by the fact that it uses the concept of event notification to structure the interactions between the entities that implement the high level protocol. More precisely, the EVA framework offers a pure publish/subscribe interaction style to manage the interactions between entities located in the same address space. As indicated in [5], the decoupling of entities in time (the interacting entities do not need to be up at the same time) as well as space (the interacting entities do not need to know each other) is a key to scalability. Unfortunately, it is hard to implement an efficient mechanism that provides a pure distributed publish/subscribe interaction style while masking the distribution aspects of the communication [5]. In the EVA framework, we have made the choice to sacrifice the space decoupling property when the publisher and some subscriber are located on different hosts. When a transmission of an event over the network is required, the designer has to identify this type event as being a remote event. Then, it has to select the most appropriate low level communication protocol. Moreover, in the case of a point to point communication, the supplier will have to identify at runtime the host where the consumer(s) is/are running. Obviously, losing the space decoupling property and allowing the designer to adapt the communication to network conditions and application requirements allow to obtain more efficient high level communication protocols. As indicated in Section 2.1.4, the switch between a publish/subscribe interaction style and an asynchronous message passing style are partially masked by the use of listeners and notifiers which are respectively EVA supplier and notifier in charge of managing network transmission.

7 Concluding Remarks

The general idea of using event-based communication has already been proposed in many distributed computational environments where decoupled and asynchronous communication is required [9, 24, 25]. EVA differs from these systems in two main aspects. Firstly, the event channel service provided by EVA only establishes how entities that are subordinated to the same component, and therefore share the same address space, interact. It is up to the application designer to define the lower level communication protocols that control the way entities that execute within different processes will interact. The framework specifies a clear and well defined way to accommodate these protocols, and also provides several standard protocol implementations. Nevertheless, designers are free to implement their own communication protocols, if required. Allowing designers to choose the most appropriate lower level communication protocol is a critical issue in the design of specialised communication protocols. Secondly, the framework offers more than an event channel service. In particular, it provides a simple and well structured programming model over which specialised communication protocols can be easily mapped. The flexibility attained by the use of EVA components is not limited to the decoupling properties of its event channel service, but includes decoupling synchronous interactions as well as providing a black box strategy to reuse both design and coding. Furthermore, by supplying developers with a suite of ready-to-use concrete Java classes, the framework allows the development of applications to be realised in a faster and less error-prone way. For instance, considering the simple example presented in Subsection 3.2, nearly 90% of all the Java byte-code required is provided by the framework.

So far, we have used EVA to implement two systems: (i) EDEN a framework for the development of reliable distributed applications supported by a group communication service - this implementation is based on the framework presented in [12]; and ii) a support for the execution of replicated processing - this implementation comprises a simple majority voting protocol and the fault-tolerant ordering protocol presented in [3]. Our experience in implementing these applications with the support of EVA, revealed that an event-based platform is well suited for implementing higher level communication protocols in a simple and flexible way. In particular, when we compare it with a layered-based platform, the following advantages can be pointed out.

- The event channel service provided by EVA offers a generic interface that allows any two entities to interact, based solely on the type of events they consume and produce.
- By not requiring events to be transfered via a predefined interaction path, the event channel service of EVA naturally implements the functionality of the short cuts required to reduce latency on layer-based applications [15]. On the other hand, isolating the production and consumption of events via an appropriate hierarchy of component entities allows, if required, an elegant and effective way of controlling the path through which an event will be routed.
- The possibility of using a component as an entity whose interface is defined by the services it provides, and the events it produces and consumes, promotes software reuse. Further, as we have shown in Subsection 3.3, passive coupling entities can be used to efficiently adapt the interfaces of entities that, originally, have not been designed to cooperate.
- Finally, the event channel service of EVA provides a transparent way to transfer events from one source to several destinations without requiring copying or forwarding operations. This type of interaction often appears in our target

applications.

We are now working on the utilisation of specialisation techniques [23] to optimise our current implementation. This will allow us to draw conclusions on the amount of overhead that the framework introduces in the latency of applications. We believe, however, that a fine tuned implementation of the framework will not introduce substantial overheads. Further, since the *critical path* on the execution of our target applications will likely comprise exclusively the interactions implemented by the framework objects, a thoroughly optimised implementation of the framework classes will frequently yield very efficient applications.

References

[1] Notification service specification, version 1.0. Object Management Group, June 2000.

[2] BHATTI, N. T., HILTUNEN, M. A., SCHLICHTING, R. D., AND CHIU, W. Coyote a system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems 16*, 4 (nov 1998).

[3] BRASILEIRO, F., EZHILCHELVAN, P., AND SPEIRS, N. TMR processing without explicit clock synchronisation. In *Proc. of the* 14^{th} *IEEE SRDS)* (Bad Neuenahr, Germany, Sep 1995), pp. 186–195.

[4] CHANDRA, T., AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43*, 2 (Mar 1996), 225–267.

[5] EUGSTER, P. T., GUERRAOUI, R., AND SVENTEK, J. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In *Proc. of the* 14^{th} *ECOOP, LNCS 1850* (Sophia Antipolis, France, June 2000), pp. 252–276.

[6] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: elements of reusable object-oriented software*. Addison Wesley Longman, Inc., USA, 1995.

[7] GARBINATO, B., AND GUERRAOUI, R. Flexible protocol composition in BAST. In *Proc. of the* 18^{th} *IEEE ICDCS* (Amsterdam, The Netherlands, May 1998), pp. 22–29.

[8] GREVE, F., HURFIN, M., RAYNAL, M., AND TRONEL, F. Primary component asynchronous group membership as an instance of a generic agreement framework. In *Proc. of the* 15^{th} *ISADS* (Dallas, Texas, March 2001), pp. 93–100.

[9] HARRISON, T. H., LEVINE, D. L., AND SCHMIDT, D. C. The design and performance of a real-time CORBA event service. In *Proc. of the 1997 ACM SIGPLAN Conference OOPSLA* (Atlanta, USA, Oct 1997), pp. 184–200.

[10] HARTKOPF, J., AND GUPTA, S. Event notifier, a pattern for event notification. *Java Report 3*, 7 (Jul 1998), 19–56.

[11] HAYDEN, M. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[12] HURFIN, M., MACÊDO, R., RAYNAL, M., AND TRONEL, F. A generic framework to solve agreement problems. In *Proc. of the* 19^{th} *IEEE SRDS* (Lausanne, Switzerland, Oct 1999), pp. 56–65.

[13] HURFIN, M., MOSTÉFAOUI, A., AND RAYNAL, M. Consensus in asynchronous systems where processes can crash and recover. In *Proc. of the* 17^{th} *IEEE Symposium on Reliable Distributed Systems (SRDS'98)* (Oct 1998), pp. 280–286.

[14] HUTCHINSON, N. C., AND PETERSON, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE TSE 17*, 1 (Janv 1991), 64–76.

[15] LIU, X., KREITZ, C., VAN RENESSE, R., HICKEY, J., HAYDEN, M., BIRMAN, K., AND CONSTABLE, R. Building reliable, high-performance communication systems from components. In *Proc. of the* 17^{th} *ACM SOSP* (Charleston, USA, Dec 1999), pp. 80–92.

[16] MALLOTH, C. P. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, École Polytechnique Fédérale de Lausanne, 1996.

[17] MISHRA, S., FEI, L., AND XING, G. Design, implementation and performance evaluation of a corba group communication service. In *Proc. of the* 29^{th} *IEEE Int. Symp. FTCS* (Madison, Wisconsin, Jun 1999), pp. 166–173.

[18] MOSER, L. E., MELLIAR-SMITH, P. M., AGARWAL, D. A., BUDHIA, R. K., AND LINGLEY-PAPADOPOULOS, C. A. Totem: a fault-tolerant multicast group communication system. *Communications of the ACM 39*, 4 (Apr 1996), 54–63.

[19] OKI, B., PFLUEGL, M., SIEGEL, A., AND SKEEN, D. The information bus: an architecture for extensible distributed systems. In *Proc. of the* 14^{th} *ACM SOSP* (Asheville, USA, Dec 1993), pp. 58–68.

[20] POWELL, D. Group communication. *Communications of the ACM 39*, 4 (Apr 1996), 50–53. Guest Editor, special issue on Group Communication.

[21] RAYNAL, M., AND TRONEL, F. Group membership failure detection: a simple protocol and its probabilistic analysis. *Distributed Systems Engineering 6*, 3 (Sep 1999), 95–102.

[22] SCHMIDT, D. C. The adaptative communication environment: An object-oriented network programming toolkit for developing communication software. In *Proc. of the* 12^{th} *Sun User Group conference* (San Francisco, California, June 1993), pp. 186–195.

[23] SCHULTZ, U., LAWALL, J., CONSEL, C., AND MULLER, G. Toward automatic specialization of java programs. In *Proc. of the* 13^{th} *ECOOP* (Lisbon, Portugal, Jun 1999), pp. 367–390.

[24] SIEGEL, J. *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc., New York, 1996, ch. CORBAservices, Part 4: Event Service.

[25] STAROVIC, G., CAHILL, V., AND TANGNEY, B. An event based object model for distributed programming. In *Proc. of the 1995 Int. Conf. OOIS* (Dublin, Ireland, Dec 1995), pp. 72–86.

[26] VAN RENESSE, R., BIRMAN, K., AND MAFFEIS, S. Horus: a flexible group communication system. *Communications of the ACM 39*, 4 (Apr 1996), 76–83.