2.1 – Architectural Support for Integration in Distributed Reactive Systems

Maarten Boasson
Quaerendo Invenietis bv
Universiteit van Amsterdam
The Netherlands
maarten@quaerendo.com

Abstract

Due to the many possible interactions with an ever changing environment, combined with stringent requirements regarding temporal behaviour, robustness, availability, and maintainability, large-scale embedded systems are very complex in their design. Coordination models offer the potential of separating functional requirements from other aspects of system design. In this paper we present a software architecture for large-scale embedded systems that incorporates an explicit coordination model. Conceptually the coordination model consists of application processes that interact through a shared data space - no direct interaction between processes is possible. Starting from this relatively simple model we derive successive refinements of the model to meet the requirements that are typical for large-scale embedded systems.

The software architecture has been applied in the development of commercially available command-and-control and traffic management systems. Experience shows that due to the very high degree of modularity and the maximal independence between modules, these systems are relatively easy to develop and integrate in an incremental way. Moreover, distribution of processes and data, fault-tolerant behaviour, graceful degradation, and dynamic reconfiguration are directly supported by the architecture.

1. Introduction

Due to the many possible interactions with an ever changing environment, combined with stringent requirements regarding temporal behaviour, robustness, availability, and maintainability, large-scale embedded systems, like traffic management, process control, and command-and-control systems, are very complex in their design. The tasks performed by these systems typically include: (1) processing of measurements obtained from the environment through sensing devices, (2) determination of model parameters describing the environment, (3) tracking discrepancies between desired state and perceived state, (4) taking corrective action, and (5) informing the operator or team of operators about the current and predicted state of affairs. All tasks are very closely related and intertwined, and particularly in large-scale systems, there is a huge number of model parameters, which are often intricately linked through numerous dependencies. It is therefore a very natural approach to design the software for such systems as a monolithic entity, in which all relevant information (deductive knowledge and actual data) is readily accessible for all the above mentioned parts.

There is, however, a strong and well-known reason to proceed differently: a software system thus conceived is very difficult to implement, and even more difficult to modify should the purpose of the system be changed, or the description of the environment be refined. Adopting a *modular* approach to design, the various functions implemented in software are separated into different modules that have some independence from each other. Such an approach - well established today as standard software engineering practice - leads to better designs, and reduces development time and the likelihood of errors.

Unfortunately, with today's highly sophisticated systems, this is still not good enough. In addition to the functional requirements of these systems, many non-functional requirements, such as a high degree of availability and robustness, distribution of the processing over a possibly wide variety of different host processors, and (on-line) adaptability and extendibility, place constraints on the design freedom that can hardly be met with current design approaches. A methodology for the design of large-scale distributed embedded systems should provide (a basis for) an integral solution for the various types of requirements. Traditional design methods based on functional decomposition are not adequate. The sound principle of modularity needs therefore to be further exploited to cover non-functional requirements as well.

Recently coordination models and languages have become an active area of research [6]. In [7] it was argued that a complete programming model consists of two separate components: the computation model and the coordination model. The computation model is used to express the basic tasks to be performed by a system, i.e. the system's functionality. The coordination model is applied to organize the functions into a coherent ensemble; it provides the means to create processes and facilitates communication. One of the greater merits of separating computation from coordination is the considerably improved modularity of a system. The computation model facilitates a traditional functional decomposition of the system, while the coordination model accomplishes a further decoupling between the functional modules in both space and time. This is exemplified by the relative success of coordination languages in the field of distributed and parallel systems.

Since the early 80's we have developed and refined a software architecture for large-scale distributed embedded systems [2], that is based on a separation between computation and coordination. Below, we first present the basic software architecture, after which we shall focus on the underlying coordination model. We demonstrate how the basic coordination model can be gradually refined to include non-functional aspects, such as distributed processing and fault-tolerance, in a modular fashion. The software architecture has been applied in the development of commercially available command-and-control, and traffic management systems. We conclude with a discussion of our experiences in the design of these systems.

2. Software architecture

A software architecture defines the organisational principle of a system in terms of types of components and possible interconnections between these components. In addition, an architecture prescribes a set of design rules and constraints governing the behaviour of components and their interaction [4]. Traditionally, software architectures have been primarily concerned with structural organisation and static interfaces. With the growing interest in coordination models, however, more emphasis is placed on the organizational aspects of behaviour and interaction.

In practice, many different software architectures are in use. Some well-known examples are the Client/Server and Blackboard architectures. Clearly, these architectures are based on different types of components - clients and servers versus knowledge sources and blackboards - and use different styles of interaction - requests from clients to servers versus writing and reading on a common blackboard.

The software architecture, named SPLICE, that we developed for distributed embedded systems basically consists of two types of components: *applications* and a *shared data space*. Applications are active, concurrently executing processes that each implement part of the system's overall functionality. Besides process creation, there is no direct interaction between applications; all communication takes place through a logically shared data space simply by reading and writing data elements. In this sense SPLICE bears strong resemblance to coordination languages and models like Linda [5], Gamma [1], and Swarm [9], where active entities are coordinated by means of a shared data space.

2.1. The shared data space

The shared data space in SPLICE is organized after the well-known relational data model. Each data element in the shared data space is associated with a unique *sort*, that defines its structure. A sort definition declares the *name* of the sort and the *record fields* the sort consists of. Each record field has a type, such as integer, real, or string; various type constructors, such as enumerated types, arrays, and nested records, are provided to build more complex types.

Sorts enable applications to distinguish between different kinds of information. A further differentiation between data elements of the same sort is made by introducing identities. As is standard in the relational data model, one or more record fields can be declared as *key* fields. Each data element in the shared data space is uniquely determined by its sort and the value of its key fields. In this way applications can unambiguously refer to specific data elements, and relationships between data elements can be explicitly represented by referring from one data element to the key fields of another.

To illustrate, we consider a simplified example taken from the domain of air traffic control. Typically a system in this domain would be concerned with various aspects about flights, such as flight plans and the progress of flights as tracked from the reports that are received from the system's surveillance radar. Hence, we define sorts *flightplan*, *report*, and *track* as indicated in Figure 1.

Sort *flightplan* declares four fields: a flight number, e.g. KL332 or AF1257, the scheduled time for departure and arrival, and the type of aircraft that carries out the flight, e.g. a Boeing 737 or an Airbus A320. By declaring the flight number as a key field, it is assumed that each flight plan is uniquely determined by its flight number.

Sort *report* contains the measurement vector of an object as returned at a specific time by the system's surveillance radar. The measurement vector typically con-

```
sort flightplan
    key flightnumber : string
    departure : time
    arrival : time
    aircraft : string

sort report
    key index : integer
    measurement : vector
    timestamp : time

sort track
    key flightnumber : string
    timestamp : time
    state : vector
```

Figure 1. Sort definitions: an example.

tains position information. A unique index is attached to be able to distinguish between different reports.

Through a correlation and identification process, the progress of individual flights is recorded in sort *track*. The state vector typically contains position and velocity information on the associated flight number, that is computed from consecutive measurements. The timestamp identifies the time at which the state vector has been last updated.

2.2. Applications

Basically, applications interact with the shared data space by writing and reading data elements. SPLICE does not provide an operation for globally deleting elements from the shared data space. Instead, data can be removed implicitly using an overwriting mechanism. This mechanism is typically used to update old data with more recent values as the system's environment evolves over time. Additionally, applications can hide data, once read, from their view. This operation enables applications to progressively traverse the shared dataspace by successive read operations. By the absence of a global delete operation, the shared dataspace in SPLICE models a dynamically changing information store, where data can only be read or written. This contrasts the view where data elements represent shared resources, that can be physically consumed by applications.

SPLICE extends an existing (sequential) programming language with coordination primitives for creating processes and for interacting with the shared dataspace. More formally, the primitives are defined as follows.

- **create**(*f*): creates a new application process from the executable file named *f*, and run it in parallel to the existing applications.

- **write**(α, *x*): inserts an element *x* of sort α into the shared data space. If an element of sort α with the same key value as *x* already exists in the shared dataspace, then the existing element is replaced by *x*.

- **read**(α, *q*, *t*): reads an element of sort α from the shared dataspace, satisfying query *q*. The query is formulated as a predicate over the record fields of sort α. In case a matching element does not exist, the operation blocks until either one becomes available or until the timeout *t* has expired. If the latter occurs, a timeout error is returned by the operation. The timeout is an optional argument: if absent the read operation simply blocks until a matching element becomes available. In case more than one matching element can be found, one is selected non-determinstically.

- **get**(α, *q*, *t*): operates identically to the read operation, except that the element returned from the shared dataspace becomes hidden from the application's view, that is, the same element cannot be read a second time by the application.

The overwriting mechanism that is used when inserting data elements into the shared dataspace potentially gives rise to conflicts. If at the same time two different applications each write a data element of the same sort and with the same key value, one element will overwrite the other in a nondeterministic order. Consequently one of the two updates will be lost. In SPLICE this type of nondeterministic behaviour is considered undesirable. The architecture therefore imposes the design constraint that for each sort at most one application shall write data elements with the same key value.

As an illustration we return to the air traffic control example from the previous section. Consider an application process that tracks the progress of flight number *n*. This application continuously reads new reports from the surveillance radar and updates the track data of flight number *n* accordingly. The application process can be defined as indicated by the code fragment in Figure 2.

The application first reads the initial track data for flight number *n* from the shared dataspace. The initial data is produced by a separate application that is responsible for track initiation. The application then enters a loop where it first reads a new report *r* from the shared dataspace. If the report correlates with the current track *t*,

```
t := get(track, flightnumber = n);
repeat
  r := get(report, true);
  if correlates(r, t) then
    update(t, r);
    write(track, t);
  end if
until terminated(t);
```

Figure 2. Coordination primitives: an example.

as expressed by the condition *correlates*(*r*, *t*), then track *t* is updated by the newly received report, using the procedure *update*(*t*, *r*). The updated track is inserted into the shared dataspace, replacing the previous track data of flight number *n*. This process is repeated until track *t* is terminated. Termination can be decided, for instance, if a track did not receive an update over a certain period of time.

3. Refinements of the architecture

The shared dataspace architecture is based on an ideal situation where many non-functional requirements, such as distribution of data and processing across a computer network, fault-tolerance, and system response times, need not be taken into account. We next discuss how, through a successive series of modular refinements, a software architecture can be derived that fully supports the development of large-scale, distributed embedded systems.

3.1. A distributed software architecture

The first aspect that we consider here is distribution of the shared data space over a network of computer systems. The basic architecture is refined by introducing two additional components. As illustrated in Figure 3, the additional components consist of *heralds* and a *communication network*.

Each application process interacts with exactly one herald. A herald embodies a local database for storing data elements, and processing facilities for handling all communication needs of the application processes. All heralds are identical and need no prior information about either the application processes or their communication requirements. Communication between heralds is established by a message passing mechanism. Messages between heralds are handled by the communication network that interconnects them. The network must support broadcasting, but should preferably also support direct addressing of heralds, and multicasting. An application process interacts with its assigned herald by means of the interaction primitives from section 2.2. The interaction with heralds is transparent with respect to the shared dataspace model: application processes continue to operate on a logically shared dataspace.

The heralds are passive servers of the application processes, but are actively involved in establishing and maintaining the required inter-herald communication. The communication needs are derived dynamically by the collection of heralds from the read and write operations that are issued by the application processes. The protocol that is used by the heralds to manage communication is based on a *subscription* paradigm that can be briefly outlined as follows.

First consider an application that performs a write operation. The data element is transferred to the application's herald, which initially stores the element into its local database, overwriting any existing element of the same sort and with the same key value.

Next consider an application that issues a read request for a given sort. Upon receipt of this request, the application's herald first checks whether this is the first request for that particular sort. If it is, the herald broadcasts the name of the sort on the network.

All other heralds, after receiving this message, register the herald that performed the broadcast as a *subscriber* to the sort carried by the message. Next each herald verifies if its local database contains any data elements of the requested sort, previously written by its application process, in which case copies of these elements are transferred to the newly subscribed herald. After this initial transfer, any subsequently written data of the requested sort will be immediately forwarded to all subscribed heralds.

Each subscribed herald stores both the initially and all subsequently transferred copies into its local database,

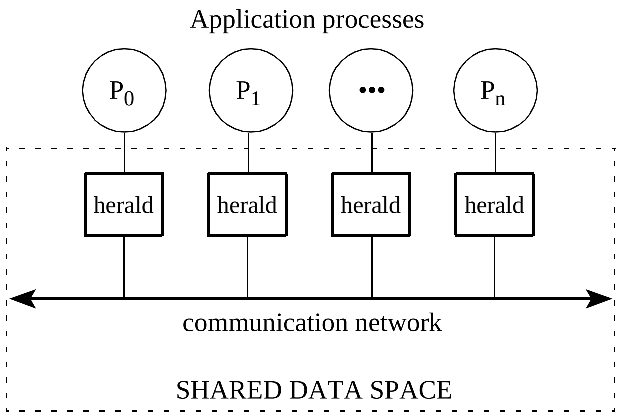

Figure 3. A distributed software architecture.

overwriting any existing data of the same sort and with the same key value. During all transfers a protocol is used that preserves the order in which data elements of the same sort have been written by an application. This mechanism in combination with the architecture's design constraint that for each sort at most one application writes data elements with the same key value, guarantees that overwrites occur in the same order with all heralds. Otherwise, communication by the heralds is performed asynchronously.

The search for data elements matching the query of a read request is performed locally by each herald. If no matching element can be found, the operation is suspended either until new data of the requested sort arrives or until the specified timeout has expired.

Execution of a get operation is handled by the heralds similarly to the read operation, except that the returned data element is removed from the herald's local database.

As a result of this protocol, the shared dataspace is selectively replicated across the heralds in the network. The local database of each herald contains data of only those sorts that are actually read or written by the application it serves. In practice the approach is viable, particularly for large-scale distributed systems, since the applications are generally interested in only a fraction of all sorts. Moreover, the communication pattern in which heralds exchange data is relatively static: it may change when the operational mode of a system changes, or in a number of circumstances in which the configuration of the system changes (such as extensions or failure recovery). Such changes to the pattern are very rare with respect to the number of actual communications using an established pattern. It is therefore beneficial from a performance point of view to maintain a subscription registration. After an initial short phase each time a new sort has been introduced, the heralds will have adapted to the new communication requirement. This knowledge is subsequently used by the heralds to distribute newly produced data to all the heralds that hold a subscription. Since subscription registration is maintained dynamically by the heralds, all changes to the system configuration will automatically lead to adaptation of the communication patterns.

Note that there is no need to group the distribution of a data element to the collection of subscribed heralds into an atomic transaction. This enables a very efficient implementation in which the produced data is distributed asynchronously and the latency between actual production and use of the data depends largely on the consuming application processes. This results in upper bounds that are acceptable for distributed embedded systems where timing requirements are of the order of milliseconds.

3.2. Temporal aspects

The shared dataspace as introduced in section 2, models a persistent store: data once written remains available to all applications until it is either overwritten by a new instance or hidden from an application's view by execution of a get operation. The persistence of data decouples applications in time. Data can be read, for instance, by an application that did not exist the moment the data was written, and conversely, the application that originally wrote the data might no longer be present when the data is actually read.

Applications in the embedded systems domain deal mostly with data instances that represent continuous quantities: data is either an observation sampled from the system's environment, or derived from such samples through a process of data association and correlation. The data itself is relatively simple in structure; there are only a few data types, and given the volatile nature of the samples, only recent values are of interest. However, samples may enter the system at very short intervals, so sufficient throughput and low latency are crucial properties. In addition, but to a lesser extent, embedded systems maintain discrete information, which is either directly related to external events or derived through qualitative reasoning from the sampled input.

This observation leads us to refine the shared dataspace to support volatile as well as persistent data. The sort definition, which basic format was introduced in section 2.1, is extended with an additional attribute that indicates whether the instances of a sort are volatile or persistent. For persistent data the semantics of the read and write operations remain unchanged. Volatile data, on the other hand, will only be visible to the collection of applications that is present at the moment the data is written. Any application that is created afterwards, will not be able to read this data.

Returning to the air traffic control example from Figure 1, the sort *report* can be classified as volatile, whereas the sorts *track* and *flightplan* are persistent. Consequently, the tracking process, as specified in Figure 2, does not receive any reports from the surveillance radar that were generated prior to its creation. After the tracking process has been created, it first gets the initial track data and then waits until the next report becomes available.

Since the initial track data is produced exactly once, the tracking process must be guaranteed to have access to it, otherwise the process might block indefinitely. This implies that the sort *track* must be persistent.

The subscription-based protocol, that manages the distribution of data in a network of computer systems, can be refined to exploit the distinction between volatile and persistent data. Since volatile data is only available to the applications that are present at the moment the data is written, no history needs to be kept. Consequently, if an application writes a data element, it is immediately forwarded to the subscribed heralds, without storing a copy in the application's local database. This optimization reduces the amount of storage that is required. Moreover, it eliminates the initial transfer of any previously written data elements, when an application performs the first read operation on a sort. This enables a newly created application to integrate into the communication pattern without initial delay, which better suits the timing characteristics that are typically associated with the processing of volatile data.

3.3. Fault-tolerance

Due to the stringent requirements on availability and safety that are typical of large-scale embedded systems, there is the need for redundancy in order to mask hardware failures during operation. Fault-tolerance in general is a very complex requirement to meet and can, of course, only be partially solved in software. In SPLICE, the heralds can be refined to provide a mechanism for fault-tolerant behaviour. The mechanism is based on both data and process replication. By making fault-tolerance a property of the software architecture, the design complexity of applications can be significantly reduced.

In this paper we only consider failing processing units, and we assume that if a processor fails, it stops executing. In particular we assume here that communication never fails indefinitely and that data does not get corrupted.

If a processing unit in the network fails, the data that is stored in this unit, will be permanently lost. The solution is to store copies of each data element across different units of failure. The subscription-based protocol described in section 3.1 already implements a replicated storage scheme, where copies of each data element are stored with the producer and each of the consumers. The basic protocol, however, is not sufficient to implement fault-tolerant data storage in general. For instance, if data elements of a specific sort have been written but not (yet) read, the elements are stored with the producer only. A similar problem occurs if the producers and consumers of a sort happen to be located on the same processing unit.

The solution is to store a copy of each data element in at least one other unit of failure. The architecture as depicted in Figure 3 is extended with an additional type of component: a persistent database. This component executes a specialized version of the subscription protocol. On start-up a persistent database broadcasts the name of each persistent sort on the network. As a result of the subscription protocol that is executed by the collection of heralds, any data element of a persistent sort that is written by an application, will be automatically forwarded to the persistent database. There can be one or more instances of the persistent database executing on different processing units, dependent on the required level of system availability. Moreover, it is possible to load two or more persistent databases with disjoint sets of sort names, leading to a distributed storage of persistent data.

When a processing unit fails, also the applications that are executed by this unit will be lost. The architecture can be refined to support both passive and active replication of applications across different processing units in the network.

Using passive replication, only one process is actually executing, while one or more back-ups are kept off-line, either in main memory or on secondary storage. When the processing unit executing the active process fails, one of the back-ups is activated. In order to be able to restore the internal state of the failed process, it is required that each passively replicated application writes a copy of its state to the shared dataspace each time the state is updated. The internal state can be represented by one or more persistent sorts. When a back-up is activated, it will first restore the current state from the shared dataspace and then continue execution.

When timing is critical, active replication of processes is often a more viable solution. In that case, multiple instances of the same application are executing in parallel, hosted by different processing units; all instances read and write data. Typically active replication is used when timing is critical and the failing component must be replaced instantaneously.

The subscription-based protocol can be refined to support active replication transparently. If a particular instance of a replicated application performs a write operation, its herald attaches a unique replication index as a

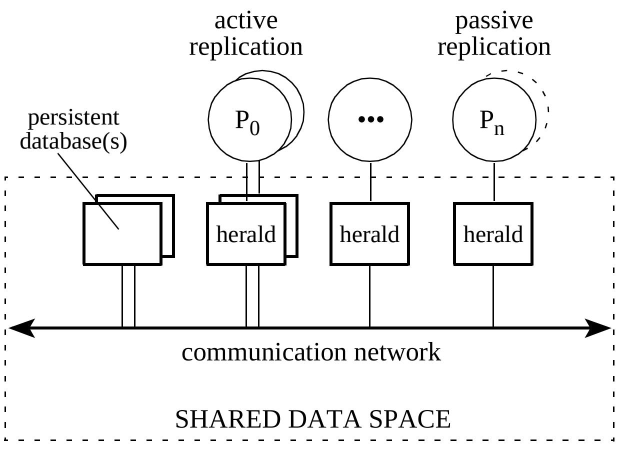

Figure 4. Supporting fault-tolerance.

key field to the data element. The index allows the subscribed heralds to distinguish between the various copies that they receive from a replicated application. Upon a read request, a herald first attempts to return a matching element having a fixed default index. When, after some appropriate time-out has expired, the requested element is still not available, a matching element with an index other than the default is returned. From that moment on it is assumed that the application corresponding to the default index has failed, and the subscription registration is updated accordingly. The index of the actually returned data element now becomes the new default.

A general overview of the distributed software architecture supporting fault-tolerance based on the various data and process replication techniques is given in Figure 4.

3.4. System Modifications and Extensions

In the embedded systems domain requirements on availability often make it necessary to support modifications and extensions while the current system remains online. There are two distinct cases to be considered.

- The upgrade is an extension to the system, introducing new applications and sorts but without further modifications to the existing system.

- The upgrade includes modification of existing applications.

Since the subscription registration is maintained dynamically by the heralds, it is obvious that the current protocol can deal with the first case without further refinements. After installing and starting a new application, it will automatically integrate.

The second case, clearly, is more difficult. One special, but important, category of modifications can be handled by a simple refinement of the heralds. Consider the problem of upgrading a system by replacing an existing application process with one that implements the same function, but using a better algorithm, leading to higher quality results. In many systems it is not possible to physically replace the current application with the new one, since this would require the system to be taken off-line.

By a refinement of the heralds it is possible to support on-line replacement of applications. If an application performs a write operation, its herald attaches an additional key field to the data element representing the application's version number. Upon a read request, a herald now first checks whether multiple versions of the requested instance are available in the local database. If this is the case, the instance having the highest version number is delivered to the application - assuming that higher numbers correspond to later releases. From that moment on, all data elements with lower version numbers, received from the same herald, are discarded. In this way an application can be dynamically upgraded, simply by starting the new version of the application, after which it will automatically integrate and replace the current version.

4. Conclusion

Due to the inherent complexity of the environment in which large-scale embedded systems operate, combined with the stringent requirements regarding temporal behaviour, availability, robustness, and maintainability, the design of these systems is an intricate task. Coordination models offer the potential of separating functional requirements from other aspects of system design. We have presented a software architecture for large-scale embedded systems that incorporates a separate coordination model. We have demonstrated how, starting from a relatively simple model based on a shared data space, the model can be successively refined to meet the requirements that are typical for this class of systems.

Over the past years SPLICE has been applied in the development of commercially available command-and-control, and traffic management systems. These systems consist of some 1000 applications running on close to 100 processors interconnected by a hybrid communication network. Experience with the development of these systems confirms that the software architecture, including all of the refinements discussed, significantly reduces the complexity of the design process [3]. Due to the high level of decoupling between processes, these systems are relatively easy to develop and integrate in an incremental way. Moreover, distribution of processes and data, fault-tolerant behaviour, graceful degradation, and dynamic reconfiguration are directly supported by the architecture.

References

[1] J.-P. Banatre, D. Le Metayer, “Programming by Multiset transformation”, Communications of the ACM, Vol. 36, No. 1, 1993, pp. 98-111.

[2] M. Boasson, “Control Systems Software”, IEEE Transactions on Automatic Control, Vol. 38, No. 7, 1993, pp. 1094-1107.

[3] M. Boasson, “Complexity may be our own fault”, IEEE Software, March 1993.

[4] M. Boasson, Software Architecture special issue (guest editor), IEEE Software, November 1995.

[5] N. Carriero, D. Gelernter, “Linda in Context”, Communications of the ACM, Vol. 32, No. 4, 1989, pp. 444-458.

[6] D. Garlan, D. Le Metayer (Eds.), “Coordination Languages and Models”, Lecture Notes in Computer Science 1282, Springer, 1997.

[7] D. Gelernter, N. Carriero, “Coordination Languages and their Significance”, Communications of the ACM, Vol. 35, No. 2, 1992, pp. 97-107.

[8] K. Jackson, M. Boasson, “The importance of good architectural style”, Proc. of the workshop of the IEEE TF on Engineering of Computer Based Systems, Tucson, 1995.

[9] G.-C. Roman, H.C. Cunningham, “Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency”, IEEE Transactions of Software Engineering, Vol. 16, No. 12, 1990, pp.1361-1373.