Criticality-Driven Component Integration in Complex Software Systems

Antonio Pecchia, Roberto Pietrantuono, and Stefano Russo

Dipartimento di Informatica e Sistemistica,
Universitá degli Studi di Napoli Federico II,
Via Claudio 21, 80125, Naples, Italy
{antonio.pecchia,roberto.pietrantuono,stefano.russo}@unina.it

Abstract. Complex software systems are commonly developed by integrating multiple, occasionally Off-The-Shelf (OTS), components. This process results into a more modular design and reduces development costs; however, it raises new dependability challenges in case of safety critical systems. Testing activities conducted during the development of the individual components might be not enough to ensure a proper safety level after the integration. The failures of the components and their impact on the overall system safety have to be assessed in critical scenarios. This paper proposes a method to support component integration in complex software systems. The method uses (i) the knowledge of the architectural dependencies among the system components, and (ii) the results of failure-modes emulation experiments, to assess both error propagation phenomena within the system and the criticality of the components in the system architecture. This information is valuable to design effective error-mitigation means and, when needed, to select the most suitable OTS item if multiple equivalent options are available. The method is applied to a real world Air Traffic Control system, developed in the context of an academic-industrial collaboration.

Keywords: Integration, Criticality Assessment, Failure-modes Emulation, Air Traffic Control.

1 Introduction

The development of complex software systems increasingly relies on the integration of existing services and components (built-in-house and, occasionally, Off-The-Shelf (OTS)), rather than on items built entirely *from scratch*. This process is commonly adopted also in critical domains, because it results, in principle, into a more modular and reliable design and it allows reducing development costs and time-to-market [1]. However, this process might raise dependability issues that are related to the integration and interactions among components [2],[3], especially in critical contexts, such as avionic and railway systems.

Software items that are reused in the design of a system are often developed either referring to a *different context*, i.e., developed for another system, or without any *specific context* in mind, i.e., the items have been developed

F. Flammini, S. Bologna, and V. Vittorini (Eds.): SAFECOMP 2011, LNCS 6894, pp. 452–466, 2011.

with the precise goal of being reused. As a result, testing activities conducted during the development of these items might be not enough to ensure a proper service during operation, because of unforeseeable interactions with the system integrating them, and the execution environment. The failures of the individual components and their impact on the overall system safety have to be carefully assessed in critical scenarios. Furthermore, the cost of the integration activities (e.g., components selection, adapters development, integration testing, design of fault tolerance mechanisms), might be even higher than the cost for developing components from scratch, especially in large and complex systems.

In this paper we present a method to support component integration in complex software systems. The method uses a model of the system at a high level of abstraction. The model encompasses the architectural components of the system, and formalizes the dependencies among them. The model is then used to drive failure-modes emulation experiments that aim at investigating error propagation phenomena within the system. Based on the error propagation paths and the error mitigation means observed in the system, the method determines a *criticality level* for each component. A criticality level represents the impact that the failures of the component have on the overall system, i.e., *the impact of its integration*. Criticalities allow engineers (i) identifying the components whose integration is potentially dangerous (these components require either more integration testing, or the design of proper fault tolerance mechanisms), (ii) comparing different equivalent components, possibly OTSs, with respect to the impact they have on the overall system safety. This information supports decisions regarding fault tolerance mechanisms, allocation of integration testing efforts, and comparison/selection of (OTS) components.

The proposed method is applied to a real world Air Traffic Control (ATC) system, developed in the context of an academic-industrial collaboration involving a world leading company, SELEX-SI, and academic partners in the COSMIC[1] project. We assess the criticality of the components integrated in the ATC system. Furthermore, because of the need of the company's system developers to select and to integrate a suitable Data Distribution Service (DDS) in the system we compare two functionally equivalent DDS platforms from the dependability perspective. Obtained results show that the experiments conducted according to the proposed method, allow identifying architectural dependencies and resources of the execution environment that impact the overall system safety, thus driving the choices taken by the project team.

The rest of the paper is organized as follows. Section 2 surveys related work in the area of dependability assessment of critical systems. Section 3 describes the proposed method and the algorithm implementing it. Section 4 and 5 describe our experience with the ATC system, and provide the results of the experimental campaign. Section 6 discusses the implications of the results on design choices.

[1] COSMIC is a three-year Italian research project aiming to create a research laboratory for the development of a open source middleware for mission critical systems.

2 Related Work

Issues related to the development by integration, such as the difficulties in controlling/testing complex interactions among components [2],[3], make dependability a significant challenge in critical scenarios. Several organizations defined standards and methodologies, such as [4], [5], [6], to support the development of dependable systems. These standards define a set of tasks and evidence to produce during the phases of the software development cycle. However, these tasks may be *time consuming*, thus neglecting the needs of current software industry. Testing and validation efforts can be driven with a preliminary knowledge of the system, in terms of criticality of its components. The standards suggest adopting *hazard analysis* and *risk assessment* techniques, such as failure modes and effects analysis (FMEA), hazard and operability (HAZOP), event tree analysis (ETA), and fault tree analysis (FTA) [8]. For example, in [9] and [10] the authors describe the hazard analysis methodology used in railway dependable systems. In [11] safety assessment processes for ATM systems have been proposed.

Several works describe approaches based on a dynamic flow graph methodology (DFM) [12], [13] to generate timed fault trees, for assessing the risk associated with dynamic behaviors. Additionally, methodologies and/or technologies for the safety assessment of real complex infrastructures and operations have been proposed. Authors in [14] present a case study to apply a goal-oriented method for car security-related hazard analysis. In [15], it has been proposed a model based on a network representation, where objects represent concepts and links represent relations. Nevertheless, this type of works do not consider the impact of the system architecture on dependability attributes.

Some issues might compromise the effectiveness of existing approaches in industrial scenarios. For example, the DFM analysis does not provide mechanisms to cope with the computational complexity of large-scale software systems. Furthermore, risk assessment is often performed by examining only faults at the interface level without considering the mitigation means included in the architecture of the system. To overcome these limitations, the proposed method adopts a system model, whose *grain* is decided by the analyst; this allows lowering the complexity of the assessment task. Furthermore, the failure-modes emulation experiments highlight error mitigation means implemented by the system.

3 Integration Strategy

In the following we describe the integration strategy. Section 3.1 provides background definitions. The assumptions for the failure-modes emulation experiments are described in Section 3.2. Section 3.3 formalizes the integration algorithm.

3.1 Background: System Model and Criticality Levels

A software system is assumed to be made by a set of *software elements*, which interact to implement the services provided by the system. Elements consist of

entities and *resources*. An **entity**, is a stateful *active* element, which interacts with resources and/or other entities. A **resource**, is stateful *passive* element, i.e., it does not interact with any other system element. Interactions can induce a state modification in the target element (*stateful* interactions), or they can leave the state of the element unchanged (*stateless* interactions).

We focus on the *dependencies* among the system elements to take into account propagation phenomena. More specifically, we assume the existence of (i) a **control dependency** between two entities A and B (or between an entity A and a resource R), if there is a direct interaction between them, and (ii) a **state dependency** between two entities A and B, if there is an interaction between A and a resource on which B performs a *stateful* interaction.

A **service** provided by the system is implemented via a sequence of interactions. Let s be such a service, with $s = 1 \dots N$ (N is the total number of services). We associate two matrix to each service s, as follows. To represent *control dependencies*, we define a matrix C_s, *(n + m)*(n + m)*, with n denoting the number of entities and m the number of resources. The element $C_{s_{ij}}$ is 1 if an interaction exists between the entity E_i and the entity (resource) E_j (R_j). Fig.2 shows an example of this type of matrix with reference to the case study. To obtain the *state dependencies* between the elements implementing a service s, we calculate a matrix S_s, as follows: (i) for each C_s, we extract a matrix Cr_s, with a value 1 only for each *entity-resource* stateful interaction, (ii) then, we sum each obtained Cr_s matrix, and transpose the resulting matrix, obtaining C_T, (iii) finally, for each service s, $C_s \ * C_T$ (rows * columns) returns S_s.

The criticality of a system element is quantified via the notion of **criticality level** (CL). The value of a CL is related to the *severity* of the effects of the element failures. Several standards for mission and safety critical systems, such as [4], [5], and [7], provide specific CL rankings. In the context of this work, without loss of generality, we consider a *generic ranking* encompassing four CLs, i.e., $1 \dots 4$, with 1 denoting the highest CL. We adopt a reverse ranking (higher the criticality, lower the CL) as in the avionic standards, because of the nature of proposed case study; however, any other choice would have been equivalent. Adopted CLs, which, again, represent the risk associated with a system function, and, in turn, with the software system element(s) implementing the function, are: `HIGH (1)`, i.e., software whose failure causes or contributes to the occurrence of a catastrophic condition, `MEDIUM (2)`, for software whose failure results in major failure conditions, `LOW (3)`, for software whose failure results in minor failure conditions, `NO criticality (4)`, for software whose failure has no effect.

3.2 Assumptions

The proposed method adopts failure-modes emulation experiments to assess the integration risk. We assume a set of failure-modes representing *how* a system element (either entity or resource) can fail [16]. In the case of *entities* we consider (i) **crash**, i.e., the entity stops providing service due to unexpected failure; (ii) **passive hang**, i.e., the entity waits indefinitely for a resource which will never be released (e.g. deadlocks) or for signals which will never be generated; (iii) **active**

hang, i.e., the entity indefinitely halts, but it keeps the system resources busy. Failures related to *return* values are not considered in this study: we assume that an interaction with correct parameters does not modify the state inconsistently and returns a correct value, i.e. a value within the expected value domain [16]. Also, we assume that the underlying network does not alter returned values (reliable channels assumption). As for *resources*, we consider the following failure-modes (i) **access denied**: the resource becomes unavailable; (ii) **read denied**: the resource is accessed, but it can not satisfy a reading request; (iii) **write denied**: the resource is accessed, but it can not satisfy a writing request; (iv) **corruption**: the content of the resource is altered.

The entity-resource model has to be tailored to the system under analysis and to the chosen grain and level of abstraction. For example, application components and OS processes may be regarded as entities and resources might be OS resources or databases. *Entities and resources have to be identified before applying the algorithm.* Furthermore, we assume that the proposed method is applied after a preliminary hazard assessment step: for each *service* the potential hazards have been identified and assigned a criticality level.

3.3 Algorithm

The integration strategy is formalized as a novel algorithm that, expanding the set of depending entities *recursively*, assigns a criticality level to the system elements involved in the provisioning of a given service. Criticality levels are represented by i) the **CL** ranking defined above, for entities, and ii) **labels** for resources indicating if they are critical or not for the system. The algorithm outputs the integration risk for each entity as a result of i) individual criticality, and ii) failure propagation paths (and intrinsic mitigation means). We report a C-like version of the algorithm, whose key steps are detailed in the following.

```
1. void RLAssignement( system Sys ){
2. EntitySet E = all the entities
3. EntitySet BorderE = border entities set;
4. ENTITY e, reader, writer, ToExpand;
5. MATRIX C,S; //dependency matrices
6. int N,M; //Number of entities and resources
7. for(e ∈ BorderE ){
8.       InitialCL = getHaResult(e);    //STEP 1: Initial definition of CLs
9.       //STEP 2: Control-CL assignment
10.      ToExpand = e;
11.      Expand (ToExpand); //function defined below
12.      //STEP 3: State-CL adjustment
13.      //For each pair of entities(e_i,e_j) set CLs according to Table 1
14       for (S_ij != 0 ∈ S){  //S_ij value is the name of resource causing the dependency
15.        readerEntity = getReaders(S_ij);
16.        writerEntity = getWriters(S_ij);
17.        if (writerEntity    ! ∈ (R.isNotRobust)){//if it is not robust to R failures
18.          AssignCL(writer)=min(reader,writer); //it pushes a CL value onto
19.                                               //the writer CL stack
20.          SetRobustEntities(S_ij,writerEntity)  //set the S_ij isNotRobust vector
21.        }}                                     //value for "writerEntity" entry
22. }
23. for (e ∈ E){    //STEP 4: Final adjustment of the CL value
24.      FinalCL(e) = min(e.CLs);
25.  }}
```

```
1. voidExpand (Entity ToExpand) {
2. RESOURCE dcR; ENTITY dcE;
3. EntitySet DCE; ResourceSet DCR;
4. //building depending entities and resources sets
5. int i = getRow(ToExpand)//get the C row corresponding to "ToExpand"
6. for (int j = 0 to N+M){
7.      if (C_ij != 0) {
8.          if (j <= N)
9.              DCE = DCE + E_j;
10.         else
11.             DCR = DCR + R_(j-N);
12.     }}
13. //evaluating robustness of "ToExpand" entity with respect to dcR failures
14. for( dcR ∈ DCR) {
15.     if (ToExpand     ∈ dcR.isNotRobust)
16. //add ToExpand entity to isNotRobust vector resource dcR
17.         SetRobustEntities(dcR,ToExpand);
18. }
19.//evaluating robustness of "ToExpand" entity with respect to dcE failures
20. for(dcE ∈ DCE){
21.         if (ToExpand! ∈ dcE.isNotRobust)
22.             AssignCL(dcE)= getCL(ToExpand)+1;  //lower its risk level
23.         else//the same risk level
24.             AssignCL(dcE)=getCL(ToExpand);
25.         ToExpand = dcE;
26.         Expand(ToExpand); //recursive call
27.  }}
```

The main types in the algorithm are: ENTITY, RESOURCE and EntitySet, i.e. a set of "Entity". Each ENTITY type has associated a stack of CL values, named "CLs", because of possible involvement in more than one service and therefore in more than one CL assignment. Both RESOURCE and ENTITY types have associated a vector of entity names, named "*isNotRobust*". For each resource R (entity E), this vector contains all the entities that are experienced as not robust to the resource R (entity E) failures itself. A set of auxiliary functions is explained by comments.

The goal is to verify (by diving into the system elements tree) if a specific element implements mitigation means to tolerate, if not, to stop, the propagation of a failure induced by other depending elements. In this case, the criticality level of the failing element can be lowered, or the element can be labeled as non-critical resource, because its failures are tolerated; otherwise, the criticality level remains unchanged. The main subsequent steps follow:

STEP 1. A starting CL is assigned to the entities directly responsible for providing a given service, i.e. the so-called *border entities*: for all identified hazards related to the given service (again, it is assumed that a preliminary hazard assessment has been performed (Section 3.2)), the minimum observed CL is assigned to the border entity.

STEP 2. Given an entity E_i, with an assigned CL value, this step aims to (i) assign a CL to the entities which it depends on (by a *control dependency*) and (ii) assess the robustness of E_i with respect to the failure of resources which it has a *control dependency* (due to reading or writing operations). The dependency relation causes a mutual effect among involved entities, so that a failure in one of them can impact the behavior of depending ones. If we denote with E_i the entity with an assigned CL, and with E_j a depending entity, in order to assign a

Table 1. CL assignment due to state dependency. s=min(CL[E_i], CL[E_j])

E_j *(write)* / E_i *(read)*	Not robust	Robust
Not Robust	**CL[E_j]=s** reading/writing critical resource	**CL unchanged** reading critical resource
Robust	**CL[E_j]=s** writing critical resource	**CL unchanged** non critical resource

CL to E_j, we consider the behavior of E_i once a failure of E_j occurs. According to the failure-modes E_i can tolerate the failure of E_j, mitigate it, or it can not be able to tolerate such a failure. If the failure of E_j is mitigated, we can lower the criticality level of E_j (i.e. increasing its ranking). Otherwise, we set CL[E_j] = CL[E_i], because E_j has to be considered as critical at least as E_i. In other words, if the failure of an entity is mitigated, this entity can be considered less critical for the overall system. In practice, we evaluate the robustness of an entity to other entities failures, through failure injections campaigns, according to the adopted failure-modes. As for *resources*, we are interested in figuring out if a resource is risky for the system, i.e., if its failures are not tolerated by the entities accessing it.

STEP 3. In the third step, we modify CLs according to the *state dependencies*. Two entities, e.g., E_i and E_j, might depend on each other through a resource R by either reading or writing access. If both entities read from R, there is no dependency between them, because they do not alter the state of the resource. Similarly, if both the entities only write on R (thus no one reads the changed state) we say that there is no dependency. A dependency exists if one of the entities writes and the other one reads the resource. In this case, if both entities have been assigned a CL, there are four possibilities shown in Table 1. A CL can be modified according to the robustness of the entities to the failures of R. If the writing entity is robust to the failures of R (e.g., if it can detect and recover from a failure by exploiting temporal and/or spatial redundancy), we do not modify the CL assigned in the previous step. Otherwise, if it does not tolerate the failures of R, i.e., some writings can be lost and this can compromise the *reading* entity, we have to consider CL[E_j] at least critical as the reading entity (i.e., CL[E_j] $\leq$ CL[E_i]).

STEP 4. In the final step, we *adjust* the CL values. Since the algorithm analyzes the system through each of the provided services individually, an entity E_i might be involved in more than one service and thus assigned more than one CL. The final CL for E_i is the minimum of observed CLs.

The output of the algorithm is a set of labels: for the entities, they indicates a CL value; as for resources, each label points out if it is a critical resource for the system or not. Furthermore, the algorithm allows achieving insights on the failure propagation paths and intrinsic mitigation means of the system.

4 Case Study

The proposed integration strategy is applied to an Air Traffic Control (ATC) system developed in the context of an academic-industrial collaboration. A preliminary experience with this system is presented in [19]. The components and the middleware layers composing the system are described in Section 4.1. In Section 4.2 we present how the proposed strategy has been applied to the system.

4.1 ATC System

The reference case study consists of a real-world ATC system. In particular, we consider a **Flight data PLan (FPL) Processor**, developed atop an open-source middleware platform, named CARDAMOM[2]. A FPL provides information, such as, the flight route, the expected trajectory of the airplane, airplane-related information, and meteorological data. The ATC system uses (i) services of the CARDAMOM platform, such as, the Load Balancer (LB), Replication (R), and System Management (SMG), and (ii) an OMG-compliant[3] Data Distribution Service (DDS) [17]. The DDS allows the components of the application to transmit the FPL instances. This is done by means of the `read` and `write` facilities provided by the DDS API, which allows to retrieve and to publish a FPL instance, respectively.

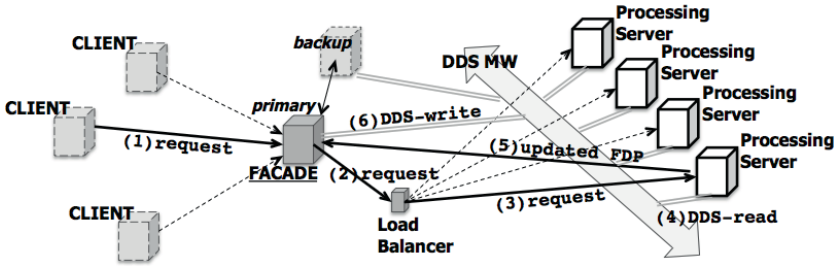

Fig. 1. Overview of the target system: FPL Processor

Fig.1 depicts the FPL Processor. It is implemented as a CORBA-based distributed objects system. The FPL Processor is composed by the **Facade** object and a pool of **Processing Servers** managed via the LB service. The system components interact as follows: the Facade object accepts FPL processing requests (i.e., insert, delete, update) supplied by external **Clients** and guarantees the data consistency by means of mutual exclusion among requests accessing the same FPL instance. The Facade redirects each allowed request to 1 out of N Processing Server, according to the *round robin* service policy. The selected

[2] CARDAMOM is a CORBA-based middleware platform providing services to support the development of software architectures for safety and mission critical systems (`http://forge.objectweb.org/projects/cardamom`)

[3] OMG specification for the Data Distribution Service, `http://www.omg.org`

server (i) retrieves the specified FPL instance from the DDS middleware (e.g., DDS MW in Fig.1) by means of the `read` facility (ii) executes request-specific computations, and (iii) returns the updated FPL instance to the Facade. The latter publishes the updated FPL instance by means of the DDS `write` facility and finalizes the request. Machines composing the testbed (Intel Pentium 4 3.2 GHz, 4 GB RAM, 1,000 Mb/s Network Interface equipped) run a RedHat Linux Enterprise 4. An Ethernet LAN interconnects these machines. As normal operation profile, Client objects invoke the services provided by the Facade with an average frequency of 50 requests per second. About 4,000 FPLs instances, each of them of 77,812 bytes, are shared with the DDS MW.

4.2 Integration Strategy

The proposed strategy is applied to the ATC system as described in Section 3. In the context of the analysis conducted in this paper, we assume the components of the ATC application (Client, Facade and processing Servers) to be *entities* and the DDS a *resource*. However, it should be noted that the grain of the model is defined by the analyst; thus, alternative models might have been chosen without the need to modify the proposed algorithm.

The **interaction matrix** is built by taking into account the interactions among the system components, as described in Section 4.1. The matrix is shown in Fig.2 (A); it has to be noted that all the services of the system exhibit the same matrix. The interaction between the Facade and the DDS is `stateful`, as the Facade writes the updated version of the FPL instance to the DDS. As a result, a *state* dependence exists between each processing Server and the Facade.

The iterative algorithm is applied as follows. The Client, i.e., the *border* entity of the reference system, is the entry point of the algorithm (Fig.2 (B) - step 1). We assume `HIGH` to be a suitable criticality level for this entity; the choice is

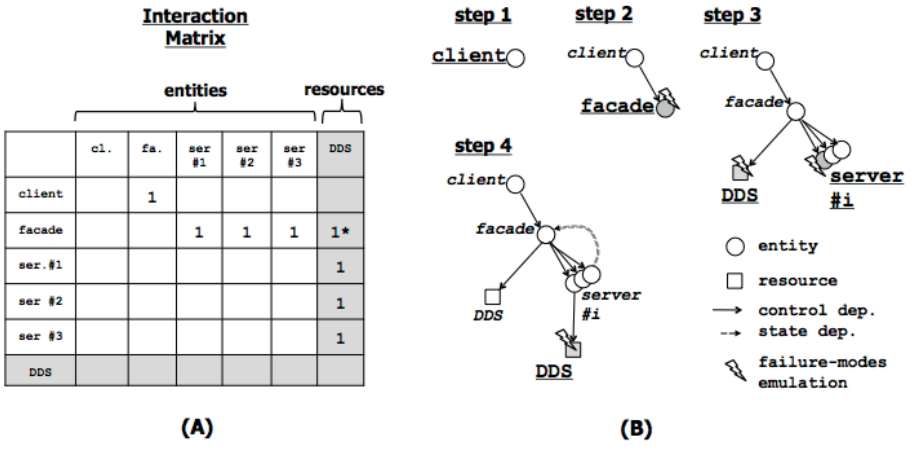

Fig. 2. Experimentation: (A) interaction matrix; (B) steps of the algorithm

Table 2. Failure-modes emulation: entities (ENT.), resources (RES.)

ENT.	**ATC component (CORBA object)**		
`crash`	the process is terminated by means of bad manipulations of an uninitialized pointer		
`active hang`	triggering of an infinite loop in the code		
`passive hang`	infinite wait on a locked semaphore		
RES.	**shared memory**	**semaphore**	**network**
`access denied`	the `vm_area_struct` related to the target shared memory is deleted from the addressing space of the process	target semaphore is deleted with `ipcrm` command-line util	network is made unavailable in two different ways (i) via the `ifconfig eth0 down` command (ii) network cable disconnection
`read denied`	bits storing the memory access policies are modified by interacting with the OS paging sub-system [18]	access permissions are modified with `semctl`; `200` is set as new value	*not meaningful in the case study*
`write denied`	bits storing the memory access policies are modified by interacting with the OS paging sub-system [18]	access permissions are modified with `semctl`; `400` is set as new value	*not meaningful in the case study*
`corrupt.`	bit-flip technique. We perform experiments by flipping a *single bit* or a *bit sequence* of increasing sizes {10, 100, 1,000, 10,000, 100,000}	target semaphore content is modified with `semctl` and the `SETVAL` flag	negligible for the case study.A dedicated LAN environment interconnects testbed machines.

reasonable in the case study since the Client represents the most external point where the service is delivered. We investigate how the failures emulated in the Facade object impact the Client in the first iteration of the algorithm (Fig.2 (B) - step 2); obtained results allow allocating a proper criticality level to the Facade object. We subsequently analyze how the Facade object behaves in case of failures emulated in the components it depends on, i.e., the processing Server and the DDS (Fig.2 (B) - step 3). Again, a criticality level is allocated to the Server and we label the DDS (at the Facade, i.e., **writer**, side) as *critical* or *non-critical* resource, according to the results of the failure-modes analysis. In the last iteration (Fig.2 (B) - step 4), we investigate how failures emulated in the DDS impact the processing Server; the DDS (at the Server, i.e., **reader**, side) is labelled as *critical* or *non-critical* resource. Failures in the Facade and Server object are emulated by triggering a faulty piece of code when a request is invoked by the Client. The adopted emulation mechanisms are described in Table 2 (ENT.). Experiment results are described in 5.1.

We analyze two OTS alternatives, coming from different vendors, as DDS middleware. The DDS middleware plays a key role in the described ATC system, both because (i) of the criticality of the domain and (ii) of the workload. For reasons of confidentiality we do not disclose the actual names of the vendors; we refer with **DDS_1** and **DDS_2** to the two DDS implementations. Both DDSs are integrated in the ATC system as follows: a shared library (`*.so`, shared object) is linked to the component that wants to use the DDS. The library relies on

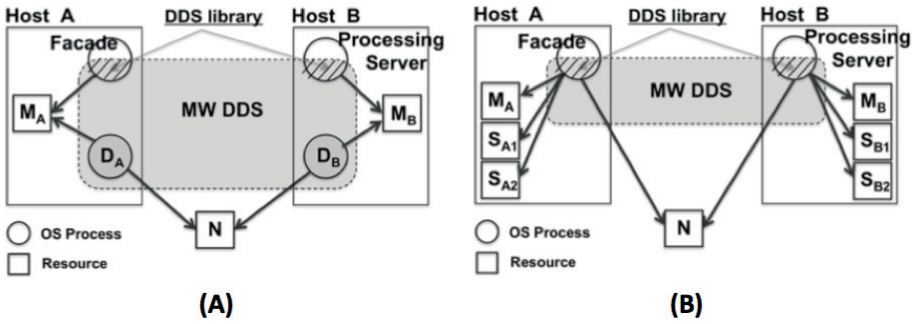

Fig. 3. Internal architecture of the DDS: (A) DDS_1, (B) DDS_2

external resources, e.g., shared memories and/or semaphores, used as support data structures (Fig.3). To assess the criticality of the DDS, we investigate how both the Facade and Server behave in case of failures occurring in the resources used by the DDS library. Table 2 (RES.) shows how these failures are emulated in the context of the Linux OS. Failures have been injected during the system operational time with kernel modules. The analysis of the criticality of the DDS is reported in Section 5.2 and 5.3, for each implementation, respectively.

5 Failure-Modes Emulation Campaign

We conduct the failure-modes emulation campaign as described in 4.2. Each emulation experiment has been repeated 10 times, in order to ensure that the results were not distorted by transitory phenomena.

5.1 ATC Components: Facade and Processing Server

The *entry point* of the integration algorithm is the Client object, whose criticality level has been assumed to be `HIGH`. We analyze the interaction **Client/Facade** beforehand; results of the failure-modes emulation experiments allow allocating a proper criticality level to the Facade. `Crash` failures of the Facade object causes the interruption of the service invoked by the Client. For example, an update request for a FPL instance does not succeed and the new data are lost, raising an exception. The emulation of `hang` failures (both `active` and `passive`) causes the interruption of the invoked service; in this case no exception is raised at the Client side. However, if other services are invoked after a hang failure, the Facade is able to process the new requests, since it is implemented as a multithreaded CORBA object. As a result, we conclude that, even if hangs are less critical than crash failures (the Facade encapsulates specific error mitigation mechanisms), a service is never able to succeed in case of failures in the Facade object. *The Facade is as critical as the Client. We assign `HIGH` as criticality level.*

We analyze the interaction **Facade/Server**. `Crash` failures of the Server object cause the request forwarded by the Facade to be lost. However, processing

Servers are organized as a load-balancing pool. For this reason, when the request is lost the Facade re-forwards the request to another server, until it is correctly executed. The emulation of `hang` failures (both `active` and `passive`) in the Server causes the interruption of the invoked service. Again, no exception is raised at the Facade side, however, the Server will be able to execute new incoming requests because of the multithreaded implementation. We conclude that the Facade is robust to crashes emulated in the Server; hangs are partially tolerated. *The processing Server is less critical than the Facade, thus it is assigned* `MEDIUM` *as criticality level.*

5.2 Analysis of the DDS_1

The DDS_1 consists of a shared library to be linked to the application, and internal middleware processes. Applicative processes (i.e., Facade and Processing Server, respectively) communicate with the DDS internal ones (D_A and D_B, *networking* processes) by means of a shared memory (Fig.3 (A)). Middleware processes are responsible for the communication among the computing nodes of a domain. Facade and processing Server interact with *shared memories* (named M_A and M_B, respectively) on both the nodes. We investigate how the failures emulated in these resources impact the correct behavior of the system.

As depicted in Fig.3 (A) the Facade object interacts with $\underline{M_A}$. As expected, the Facade crashes with a "`segmentation fault`" message in case of an `access`, `read`, or `write denied`. This is due to the nature of the Linux OS paging subsystem. The `corruption` of M_A has different consequences, depending on the modified bits. In particular, the Facade enters a hang state if the corruption affects lowest M_A bits, a crash one, otherwise. The Processing Server interacts with $\underline{M_B}$. It crashes with a "`segmentation fault`" message in case of an `access`, `read`, or `write denied`. Apparently, the `corruption` of M_B does not make the Processing Server to hang or crash; however, after the emulation experiment, the updated versions of FPL instances are not delivered to the server anymore. Furthermore, no error notifications are returned.

We conclude that *failures of M_A and M_B always compromise the mission of the ATC system.* M_A and M_B are *critical* resources at the *writer* and *reader* side, respectively, since the DDS library, integrated in the ATC components, does not encapsulate suitable mitigation means to tolerate injected failures.

5.3 Analysis of the DDS_2

The DDS_2 exhibits a different architecture (Fig.3 (B)). All the code of the DDS is mapped into the application processes in the form of a shared library. As a result, Facade and processing Servers interact *directly* with *shared memories*, *semaphores* and the *network*. Let M_A, S_{A1}, S_{A2} and M_B, S_{B1}, S_{B2} be shared memories and semaphores at Facade and Processing Server side, respectively. Let N be the *network*. Fig.3 (B) depicts the interactions to transmit data, i.e., FPLs, between two computing nodes.

The Facade process relies on $\underline{M_A}$, $\underline{S_{A1}}$ and $\underline{S_{A2}}$. As we expected, the Facade object crashes with a "`segmentation fault`" message in case of an `access`,

`read`, or `write denied`. The `corruption` of M_A does not compromise both the behavior of the Facade and data transmission (i.e., each subsequent DDS `write` invocation correctly succeeds). Furthermore, the improper modification of the shared memory content is notified with the following message:

"*`*_Transport_Shmem_attach_writer: incompatible shared memory segment found. All applications using * must use compatible shared memory protocols`*". This warning message is triggered by the DDS library every 10 seconds and it is printed on the console of the Facade. We can conclude that *failures of M_A do not necessarily compromise the mission of the ATC system.* In this case, the library of the DDS_2 encapsulates mitigation mechanisms that tolerate the corruption and notify the faulty state of the resource. $\underline{S_{A1}}$ and $\underline{S_{A2}}$ `access/read denied`, and `corruption` do not affect Facade operations. The `write denied` emulated on both the semaphores is notified with the following messages: "*`* Mutex_lock: OS semop() failure error OXD. *_send:!take semaphore`*", and: "*`* Mutex_ive: OS semctl() failure error OXD. *_send: !give semaphore.`*", respectively. The warning messages are triggered by the DDS library every 10 seconds and are printed on the Facade console. We conclude that failures emulated on S_{A1} *and* S_{A2} *do not compromise the mission of the ATC system.* DDS_2 tolerates and notifies emulated failures: thus, semaphores are not critical resources.

Processing Server uses $\underline{M_B}$, $\underline{S_{B1}}$ and $\underline{S_{B2}}$. Failure emulation provides findings similar to the Facade. DDS_2 tolerates, and occasionally notifies, emulated failures. Facade and processing Server communicate through $\underline{N}$. When we emulate network unavailability with any of the proposed mechanisms, updated FPL instances are lost. However, both processes do not exhibit any explicit notification. Communication between the nodes is restored when the network is resumed. *N unavailability compromises the mission of the system*, however DDS_2 is robust to *transient* failures of the network.

6 Design Implications and Lessons Learnt

Experiments show that the Facade is the most critical object in the ATC system. Failures of different types occurring in the Facade are not tolerated by the Client; furthermore, a `crash` of the Facade object compromises the mission of the system *as a whole*. On the other hand, the processing Server is not particularly critical. The proposed approach highlights that the system encapsulates error mitigation means to tolerate the failures occurring in the processing Server. As a result, most of the testing/validation efforts should be devoted to the Facade object. Alternatively, additional mitigation means might be included in the system to reduce the criticality of the Facade, considering the observed failure-modes. In general, *the overall safety level of a system depends on the nature of the dependencies among its components.* Thus, such dependencies should always be analyzed as showed. Indeed, the architecture of the system and design choices affect the ability of the system to react and to mitigate the failures.

The analysis of the two DDSs reveals that different implementations of the same service, i.e., the data distribution, affect the ability of tolerating failures. We observed that both DDSs use **shared memories**. In DDS_1 they have a *critical* role. Communications occur via shared memory both at the Facade and Processing Server. In other words, it is a *single point of failure*, since each emulated failure compromises data transmission. DDS_2 uses shared memories too. Anyway, in this case they provide only support facilities. Their corruption does not compromise data transmission. DDS_1 does not use **semaphores**, thus avoiding the introduction of new potential failure sources. However, they do not represent an actual dependability threat in DDS_2. Even if they are used to access resources, their failures do not compromise the service completion. Again, the same resource, e.g., the shared memory provided by the Linux OS, is critical for the first DDS implementation, but not for the other. This enforces that, in general, *distinct OTS implementations might result in different fault tolerance features with respect to the execution environment; these features have therefore to be assessed, in order to make convenient choices* (e.g., choosing a different OTS, or implement fault tolerance mechanisms).

Despite the different features of the DDS, the choice of a less dependable implementation does not affect the criticality levels of the components in the proposed case study. As discussed, a *state dependence* exists between the processing Server and the Facade. A state dependence might modify the criticality level of the writer entity, when it is not robust to the failures of the resource (see Table 1). However, the Facade, i.e., the writer, is already more critical than the processing Server, thus the criticality of these two components does not change due to the dependence. We observe that, in general, *both (i) the fault tolerance capabilities of the OTS, and (ii) the criticality of the components of the system that use the OTS are worth to be assessed, since the choice of a specific OTS component depends on both these aspects.*

In the future, we will extend the integration approach to other domains and applications. We aim to compare in terms of dependability, other than different architectures, different execution environments. Moreover, we will extend the approach by considering different kinds of OS/middleware resources, as well as other failure-modes, in order to increase the accuracy of the experimentation. One of our goals is to investigate further failure emulation techniques to deal with software, coming from different suppliers, for which vendor does not share the code and does not provide practical support for the integration (i.e., a failure emulation completely transparent to the OTS internals). This will help establishing a common procedure, *independently* from the OTS suppliers.

Acknowledgment. This work has ben partially supported by the project "CRITICAL Software Technology for an Evolutionary Partnership" (CRITICAL-STEP, `http://www.criticalstep.eu`), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 230672, within the context of the EU Seventh Framework Programme (FP7) and by the Italian Ministry for Education, University, and Research (MIUR) in the framework of the Project of National Research Interest (PRIN) "DOTS-LCCI: Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures".

References

1. Hammet, R.: Flight-Critical Distributed Systems: Design Considerations. IEEE AESS Systems Magazines, 30–36 (2003)
2. Weyuker, E.J.: Testing Component-Based Software: A Cautionary Tale. IEEE Software 15(5), 54–59 (1998)
3. Moraes, R.L.O., Durães, J., Barbosa, R., Martins, E., Madeira, H.: Experimental Risk Assessment and Comparison Using Software
4. CENELEC: EN 50126 Railways Applications. The specification and demonstration of Reliability,Availability, Maintainability and Safety (RAMS)
5. DO-178B/ED12B Software consideration in airborne systems and equipment certification. RTCA and EUROCAE (December 1992)
6. SAF.ET1.ST03.1000-MAN-01. Air Navigation System Safety Assessment Methodology (v2-0). EUROCONTROL EATMP Safety Management (April 2004)
7. Functional safety and IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems. Produced by IEC/SC65A/WG14, The working group responsible for guidance on IEC 61508 (September 2005)
8. Storey, N.: Safety-Critical Computer Systems. Pearson and Prentice Hall (1996)
9. Hassami, A.G., Foord, A.G.: Systems safety-a real example (European rail traffic management system, ERTMS). In: Proc. of the Second IEEE International Conference on Human Interfaces in Control Rooms, Cockpits and Command Centres, pp. 327–334 (2001)
10. Pasquale, T., Rosaria, E., Pietro, M., Antonio, O.: Hazard analysis of complex distributed railway systems. In: Proc. of the 22nd IEEE International Symposium on Reliable Distributed Systems (SRDS 2003), pp. 283–292 (October 2003)
11. Mana, P., De Redet, J.M., Fowler, D.: Assurance Levels for ATM elements: Human (HAL), Operational Procedure (PAL), Software (SWAL). In: Proc. of the 2nd IEEE Int. Conference on Institution of Engineering and Technology, pp. 13–19 (October 2007)
12. Garrett, C., Apostolakis, G.: Automated hazard analysis of digital control systems. Reliability Engineering and System Safety 77, 1–17 (2002)
13. Garrett, C., Guarro, S., Apostolakis, G.: The Dynamic Flowgraph Methodology for Assessing the Dependability of Embedded Software Systems. IEEE Trans. on Syst., Man, and Cybern. 25(5), 824–840 (1995)
14. Supakkul, S., Lawrence, C.: Applying a Goal-Oriented Method for Hazard Analysis: A Case Study. In: Proc. of the 4th International Conference on Software Engineering Research, Management and Applications (SERA 2006), pp. 22–30 (August 2006)
15. Hewett, R.: Assessment of Software Risks with Model-Based Reasoning. In: Proc. of IEEE Inter. Conf. on Systems, Man and Cybernetics, vol. 4, pp. 3238–3243 (2005)
16. Powell, D.: Failure Mode Assumptions and Assumption Coverage. In: Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing, FTCS 1992 (1992)
17. Pardo-Castellote, G.: OMG data-distribution service: Architectural overview. In: Proc. of the IEEE ICDCS Workshops, pp. 200–206 (2003)
18. Rubini, A., Corbet, J.: Linux Device Drivers, 2nd edn. O'Reilly, Sebastopol (2001)
19. Cotroneo, D., Pecchia, A., Pietrantuono, R., Russo, S.: A failure analysis of data distribution middleware in a mission-critical system for air traffic control. In: Proc. of the 4th ACM Int'l Workshop on Middleware for Service Oriented Computing, pp. 25–30 (2009)