Reconfiguration Service for Publish/Subscribe Middleware

Bogumil Zieba[1], Maurice Glandrup[1], Marten van Sinderen[2], Maarten Wegdam[2]

[1] Thales Nederland B.V., Haaksbergerstraat 49, Hengelo (O) 7550 GD, The Netherlands
{bogumil.zieba, maurice.glandrup}@nl.thalesgroup.com
http://www.thales-nederland.nl

[2]University of Twente, Department of Computer Science, Centre for Telematics and Information Technology,
PO Box 217, 7500 AE, Enschede, The Netherlands
{m. vansinderen, m.wegdam}@utwente.nl

Abstract. Mission-critical, distributed systems are often designed as a set of distributed, components that interact using publish/subscribe middleware. Currently, in these systems, software components are usually statically allocated to the nodes to fulfil predictability, reliability requirements. However, a static allocation of components has major drawbacks, e.g. the need for quantification of the expenditure of resources to prevent a lack of resources during run-time. A dynamic allocation diminishes the drawbacks of a static allocation by reallocating components during system run-time. The process of dynamic reallocation is considered as a reconfiguration of the system, which can be implemented as an additional functionality of the middleware. In this paper, we propose a new dynamic reconfiguration service for a publish/subscribe middleware that enables dynamic reallocation of components in order to achieve predictable and reliable system behaviour and fulfil deployment requirements. We have built a prototype that validates our research.

1. Introduction

The context of a research is distributed, mission-critical systems, where the functioning of an organization or success of a carried mission depends on the predictable and reliable system operation. Typical application of mission-critical systems can be in the field of energy management and air traffic control systems. The critical character of such systems introduces high quality requirements (ultra-quality [6]) for these systems. To fulfil the performance requirements, software components are usually statically allocated to computer nodes. The static allocation guarantees that resources required by components are available during run-time of the system. The amount of reserved resources is predicted for the worst expenditure of resources scenario [21]. However, static allocation has major drawbacks, which became a motivation for our research. The most important drawbacks are:

- *Redundant resource reservation*. Resources are reserved for the worst resource expenditure scenario, which introduces an inefficient resource usage during a sustained system operation.
- *Quantification of the resource expenditure*. Reservation of resources requires the quantification of the peak resource expenditure in the system design phase, which requires complex quantitative modelling of the system and its usage.
- *Shared resources*. During design and deployment the static allocation method requires knowledge on the usage of the shared resources by other components.
- *Failures.* Since the allocation of resources is fixed, it is not possible to change the resource allocation to recover from resource failures [9].

A dynamic reconfiguration (allocation) diminishes these drawbacks of the static allocation. The objective of the dynamic reconfiguration is to allow a system to evolve incrementally from one configuration to another at run-time, as opposed to at design-time, while introducing little (or ideally no) impact on the system's execution. In this way, systems do not have to be taken off-line, rebooted or restarted to accommodate changes [9]. In the dynamic allocation approach components are reallocated to nodes at run-

time. Information about components expenditure of resources and the usage of shared resources are fetched at run-time. The dynamic reconfiguration is an intrusive process, which consumes additional resources. It may bring in delays in messages exchange. It should introduce as little overhead as possible (ideally no overhead at all) on the system execution. This overhead is referred to the direct costs (which overheads usually CPU time, memory, storage space and communication bandwidth) and indirect costs (reallocation introduces some delay in the system) of reconfiguration process [13]. An additional concern of the dynamic reconfiguration is correctness preservation. The reconfiguration must assure that the system parts that interact with entities under reconfiguration do not fail because of reconfiguration [9]. Therefore it requires performing some studies on the trade-off between the cost of the reconfiguration and provided benefits under assurance of the correctness preservation.

The major contribution of this paper is the application of the dynamic reconfiguration concept (for example as presented in work [9]) to the publish/subscribe middleware that dynamically allocates components to nodes. Previous work on reconfiguration in a p/s communication model considers reconfiguration in terms of changes in the topology of the network [10]. However, we define the reconfiguration in terms of dynamic reallocation of components in a fixed topology of the network. We argue that reconfiguration services for connection-oriented middleware (e.g. CORBA Component Model) are not sufficient for the p/s system middleware because of a different computational model. In the connection-oriented model, objects are linked by bindings through which interactions occur [5], which is different from the p/s model, where objects are autonomous and decoupled.

In this paper, we discuss an approach for solving the drawbacks of the static allocation. The implementation of this approach is the reconfiguration service. Measurements from the prototype validate the approach.

The rest of the paper is structured as follows. Section 2 presents an overview of p/s middleware, and the p/s communication model. Section 3 defines requirements for the reconfiguration service. Section 4 presents our approach for dynamic allocation for the p/s middleware. Section 5 discusses consistency preservation during in the reconfiguration in p/s middleware. In section 6 we describe an architecture design of the reconfiguration service. Section 7 describes the prototype and measurements from the prototype. Section 8 discusses related work. Finally, conclusions are presented in section 9.
The plans for the future work are discussed in the section 10.

2. Overview of Publish/Subscribe Middleware

Middleware is reusable software that resides between applications and the underlying operating systems, network protocol stacks, and hardware [14]. Middleware's primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure to coordinate how parts of applications are connected and how they interoperate. Middleware focuses especially on issues that emerge when such programs are used across physically separated platforms [15].

2.1. High Level Model of Publish/Subscribe Middleware

The Publish/subscribe (p/s) middleware uses an interaction model that consists of information providers (publishers), which publish events to the system, and information consumers (subscribers), which subscribe to

the events of interest within the system [1]. A participant may simultaneously publish and subscribe to information [2]. A communication among participates is asynchronous provided by an infrastructure – notification service (figure 1).

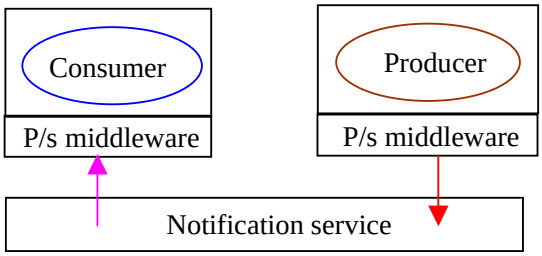

Fig. 1. Publish/subscribe middleware system infrastructure

The p/s middleware ensures the timely notification of events to interested subscribers. An event can be seen as a special message sent by an information provider and implicitly addressed to the set of information consumers, which issued a subscription that matches the event (topic) [1]. P/s middleware directly reflects the intrinsic behaviour of information-driven system because the producer of information initiates the communication among components [4]. OMG Data-Distribution Service (DDS) is an emerging specification for publish/subscribe middleware [11].

2.2. Publish/Subscribe Communication Model

In the p/s communication model the overall distributed system is composed of communicating components. Each of them is a separate process running in a separate address space possibly on different computers. The p/s model is best suited for data-centric exchange, where applications publish (supply or stream) "data" which typically requires a lot of computations, and is less suitable for distributed systems that communicate using remote procedure calls. The following particularities of the communication model impact for the overall design decision of the reconfiguration service:

1. High level of decoupling among publishers and subscribers. In [19] three dimensions of decoupling are introduced:
   - Space decoupling (that captures the fact that interacting parties do not need to known each other);
   - Time decoupling (that captures the fact that parties do not need to be actively participating in the interaction at the same time); this capability of p/s middleware is specified in [11] as the DURABILITY QoS. It allows assigning to samples of data PERSISTENT property. This data is then kept on permanent storage by p/s infrastructure and are delivered to the late-subscriber (subscriber, which was not instantiated during data publication) [11].
   - Flow decoupling (that captures the asynchrony of the model).
2. Global data space. P/s communication model creates the illusion of a shared "global data space" populated by data that components in distributed nodes can access via simple read write operation. In reality, the data does not really "live" in any one computer's address space. Rather, it lives in the local caches of all applications that have an interest in it. The local cache at a single node is shared by many components. Here is where the publish/subscribe aspect becomes key [20].

3. Requirements for Dynamic Reconfiguration in Publish/Subscribe Middleware

As stated before, the context of the research is mission-critical, distributed systems, in which reliability and predictability are fundamental

requirements. Hence, the objective of the dynamic reconfiguration is to react against events, that could threaten a reliable and predictable system behaviour. The general goal of the reconfiguration is to ensure the reliability of the system rather than optimise it in terms of e.g. execution time minimisation. We have defined the following requirements for the reconfiguration:

(Req.1) Components deployment requirements shall be fulfilled by the p/s middleware. For example the component shall be run on the machine directly connected to a certain device e.g. GPS (Global Positioning System).

(Req.2) Avoid and prevent resources of computing nodes from an overload[1]. The quantification and resource reservation in the static allocation approach ensure that unpredictable events, e.g. node overload, do not occur. In order to prevent system from the overload, the dynamic reconfiguration shall reallocate components, based on measurement of run-time system load.

(Req.3) Provide reliability of components (fault tolerance). The reconfiguration shall detect a component failure. In the case of failure, component shall be re-instantiated and reallocated at the shortest possible time.

An accomplishment of the first and second requirement ensures components predictable behaviour; rather an accomplishment of the third requirement ensures reliable system behaviour.

4. Approach

We define a system configuration as a physical allocation of all the components in the system. This configuration can be changed (reconfigured) by performing following operations: component migration, starting, and stopping. An example of a system reconfiguration is shown at figure 2.

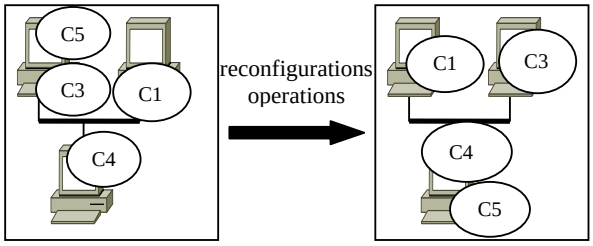

Fig. 2. Example of evolving from system configuration n to n+1

At figure 2, the system configuration *n* has been changed to configuration *n+1* by migration of components *C1*, *C3*, and *C5*.

In our approach we assumed one-to-one mapping between a component and a process, what brings in major simplifications and benefits. It allows for associating the resources needs of an individual component and with those of a process. Operating system provides built-in system calls for the measurement of process expenditure of resources. Hence, the one-to-one association between process and component highly simplifies measurements of components expenditure of resource. This design choice of the approach and p/s communication model enables us to use process placement allocation algorithms for the middleware and bridging the gap between the research done in process placement ([7]), and the reconfigurable middleware ([9]). Historically most work done in the area of dynamic resource allocation is about assigning processes to nodes. However this process-based research was not directly applied to middleware, because of following reasons:

– A process can contain a lot of components, making this a too coarse grained solution [9];
– Process migration is expensive in terms of process state that has to be transferred. Whole process state consists of thread

[1] Overload is a condition that a need for resources exceeds devices capabilities, causing undesirable consequences.

context, stacks, text region, data region static memory, dynamic region and system state.

The common design pattern used in the p/s component is "check-pointing". Components periodically or state change-driven save their state to the middleware database (explained before as the "global data space"). This capability brings in major benefits in the components reliability and migrations. When component crashes, it can be restored with the state before failure. The storing state capability highly simplifies the component migration (change of a physical location). Component migration can be limited to the following operations: saving state to the "global data space", stopping the process that embeds component, start it at an another node and load the system state.

For a component to be able to properly checkpoint its state to the p/s middleware, the component should be in a so-called safe-state. Safe state means that the component does not accept new transactions, does not initiate new transactions and any transactions it has already initiated have completed [9]. Especially for multi-threaded components, reaching a safe state is not trivial. If check-pointing takes place when a component is not in a safe state, there might be some state inconsistencies for the migrated component, e.g., because some piece of data in the data stream can be lost. In this paper we do not further address the issue of reaching safe state, since for the type of data-centric systems we are considering the state inconsistencies that might occur are minor and of little consequence. However, this depends on the characteristics of the specific system, and we will address this issue in future work.

The supplement of the approach is the definition of the deployment requirement issued by components. Each component specifies its deployment requirement to p/s system that shall be fulfilled in order to operate in predictable and reliable way.

5. Consistency Preservation During Reconfiguration in Publish/Subscribe Middleware

For distributed systems in general, and thus including for middleware-based systems consistency preservation during reconfiguration is a major issue. A system can become useless in case the preservation consistency is ignored. The system under reconfiguration must be left in a "correct" state after reconfiguration. In order to support the notion of correctness of a distributed system, three aspects of correctness requirements are identified [12]:

1. The system satisfies its structural integrity requirements. It constraints the structures of system in terms of the relationship between and the ways in which these component might be put together. In terms of CORBA it is satisfying the interface definition of the original object, and reference to new reconfigured object
2. The components in the system are in mutually consistent states. Each interaction (means by which components can effect each other's state) between them, on competition, results in a transition between well defined and consistent stated for the parts involved.
3. The application state invariants are predicates involving the state of (a subset of) the entities in a system. The preservation of safety and liveness properties of a system depends on the satisfaction of these invariants

A major contribution of this paper is the observation that the particularity of the p/s communication model provides an automatic preservation of component state consistency during the reconfiguration. The proposed reconfiguration concept does not require implementing additional functionality ("design hooks"), except saving/loading state, from the components in order to preserve "correctness" after the reconfiguration.

The space-decoupling paradigm ensures that the structural integrity requirement is not an issue for the p/s communication model. The p/s middleware changes a characteristic of a communication among software components. This communication has an indirect character, decoupled by the middleware infrastructure, contrary to the connection-oriented communication.

The time decoupling and the global data space capability ensure mutually consistent states. For example, let us consider the scenario, in which the component is migrated from one node to another during ongoing interaction. Firstly, data are delivered to the local cache of node, where a component was localized before the migration. After the migration, component is localized on another node and expresses interest in the same type of data. Local caches of two nodes (nodes before and after migration) are automatically synchronized by the p/s infrastructure. The synchronization of local caches in the "global data space" provides consistency preservation during the migration of components. One of factors influencing duration of the reconfiguration is time of the synchronization of local caches. We observed, that this time depends on the volume of data stored in the local caches. Unfortunately, at this stage of the research we cannot present any quantitative measure. We expect that the state of components to be relatively small and synchronization process shall not introduce major delays in the 'data centric' applications.

We recognize the synchronization of local caches as the important issue in the reconfiguration for p/s middleware and address it for the future consideration.

Allowing stateful components to save/load their state to/from to the "global data space" ensures the application state invariants requirement. State of the components can be saved/restored in any moment at other nodes.

This analysis of data-centric paradigm leads to the conclusion that high level decoupling among publishers and subscribers and global data space ensures automatically "correctness" after the reconfiguration.

6. High-level Architecture of Reconfiguration Service

Based on the dynamic reconfiguration approach as explained in the previous section, we designed the high-level architecture of the reconfiguration service for the p/s middleware (see figure 3).

We use the "Agent-Manager" pattern used in Simple Network Management Protocol (SNMP). In this pattern Agent is located on every node. It monitors and manages node, on which is located. High-level management is performed by the Manager, which is physically and logically centralized. Centralized approach has drawbacks of potential weak scalability and single-point-of-failure issues. In our approach a central implementation of the Manager is sufficient, because only control flows of data are centralized, and data flow is distributed (by p/s middleware system). The amount of control information is limited. We anticipate possibility of the distributed implementation of the Manager for a large-scale distributed system in a case of scalability problems.

The architectures consists of the following logical parts:

- *Reconfiguration Manager* – allocation algorithm, designed as a (logically) single, central, decision-making and information-storing component;
- *Reconfiguration Agents* – reconfiguration functionality located on every computing node, taking part in the p/s communication, managing and monitoring nodes and components.
- *P/s components* – components managed by reconfiguration service

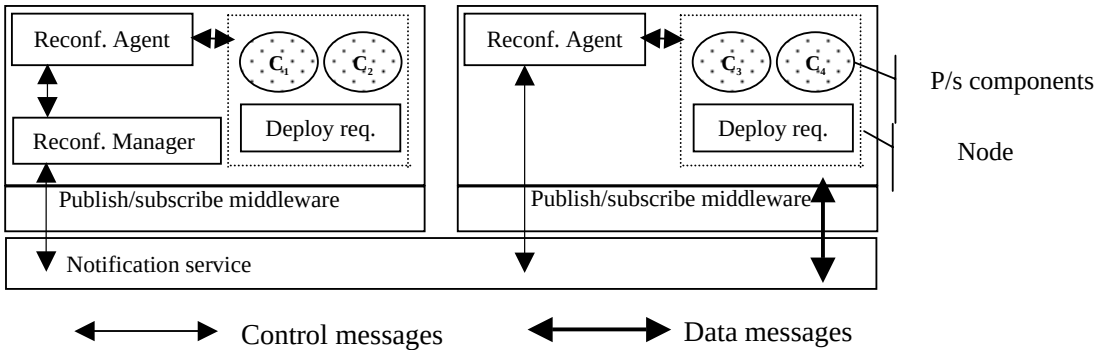

Fig. 3. The architecture of the reconfiguration service

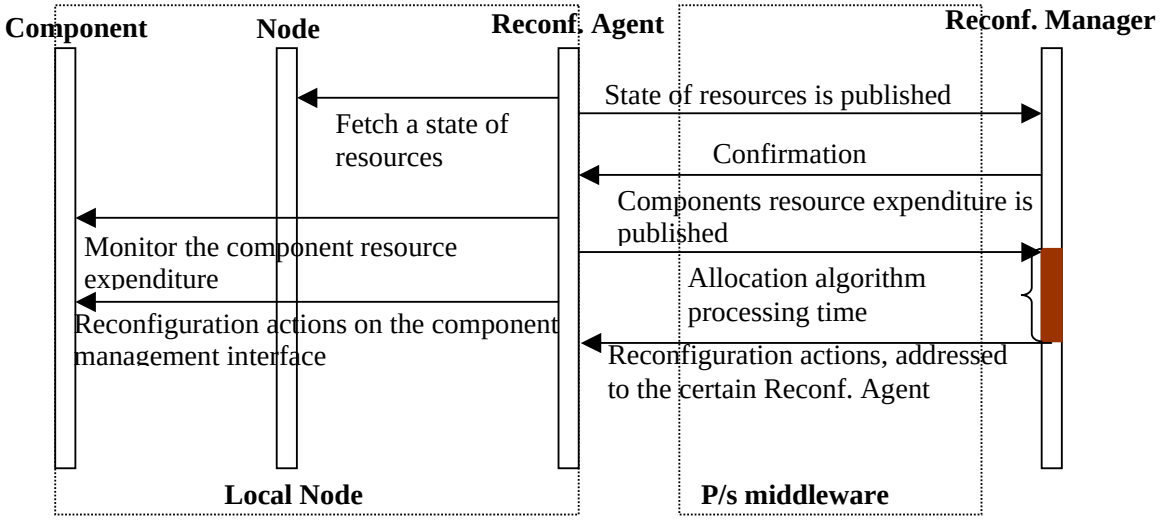

Fig. 4. The reconfiguration service interaction diagram

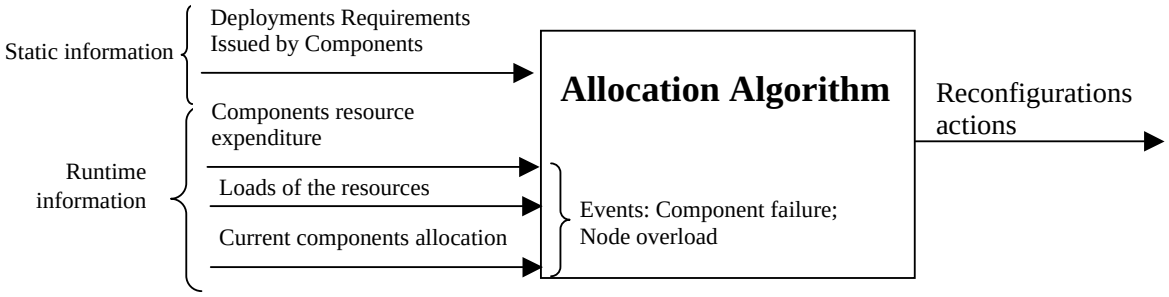

Fig. 5. Data flow in the allocation algorithm

– *Managed nodes* – computing nodes managed by the reconfiguration service.

Each component provides deployment requirements in a form of XML file and reconfiguration management interface. Through this interface the Reconfiguration Agent performs reconfiguration actions on components, such as components stopping, starting etc.

The interaction diagram of the reconfiguration service is presented at the figure 4.

Reconfiguration Agent controls, monitors components on the node it is located, and fetches state of the computing node resources, e.g., CPU load. The periodic monitoring of components ensures detection of component failures. This monitoring data is periodically sent to the Reconfiguration Manager. Based on this data, the Reconfiguration Manager detects occurrence of the following events:

– Node overload (example of the node overload definition is presented in section 7.2)
– Component failure. When the Reconfiguration Manager does not receive components expenditure of resources from the Reconfiguration Agent, it assumes the component failure.

When one of the previously defined events is detected, a new components allocation is determined. The Reconfiguration Manager implements an allocation algorithm (see figure 5 for input/output data flow), which computes new components allocation. This is determined by: static information (defined at design time) e.g. components deployment requirements, and dynamically fetched information (at runtime) e.g. nodes loads.

In order to achieve a new allocation following reconfigurations actions are available: a service migration (process of moving a service execution location from one computing node to another), a service stopping, a service starting.

The Reconfiguration Manager sends messages addressed to Reconfiguration Agents containing an instruction of the reconfiguration actions (e.g. migration), and components name.

Before Reconfiguration Agent performs any action, it requires stateful components to save/load their internal state (component context) to/from the middleware infrastructure.

The advantage of the current architecture is a capability of loading different allocation algorithms to the service. Predictability and reliability of the service are highly depended on the efficiency of an allocation algorithm. The taken approach allows using process placement allocation algorithms in the service. Subject of the allocation algorithm is out of the scope of this paper.

7. Prototype

A prototype validates the concept of the reconfiguration service for the p/s middleware. It has been built based on SPLICE[2], DDS compliant p/s middleware implementation. The prototype is evaluated in terms of fulfilling previously stated requirements, and the overall overhead of service.

Measurements were conducted in a test environment consisting of 3 computing nodes connected through computer network (Ethernet technology). Each of computing nodes had the following hardware configuration: processor Pentium 4, 2260 MHz (4482 Bogomips), 512 MB of RAM. The prototype service was developed for Linux platform, in C++.

7.1. Fulfil Deployment Requirements Issued by Components

Each component issues the required deployment requirements in the form of XML file. This file describes: component name (<name>), command line to start it

[2] SPLICE is Thales Naval Nederland proprietary publish/subscribe middleware.

(<commandlinestart>), operating system for which it was developed (<operatingsystem>), type of a computing node, on which it preferable operates (<type>), priority (<priority>), name of other component which shall not/shall preferable operate on the same node (<dispersion>, <preference>), whether it can be migrated (<migratable>), whether it can be replicated (<replicable>), allocation preference according to which it shall be allocated e.g. memory, cpu or both (<allocationpreference>). The example of this file is presented in table 1:

```
<component>
  <name>RadarSim</name>
<commandlinestart>/home/bz/RadarSim
</commandlinestart>
  <operatingsystem>Linux</operatingsystem>
  <type>processing</type>
  <priority>1</priority>
   <dispersion> TrackManager</dispersion >
  <preference >DisplayRadar</preference >
  <migratable>true</migratable>
  <replicable>false</replicable>
  <allocationpreference>
cpu</allocationpreference >
  </ component >
```

Table 1. Example of XML file, containing issued deployment requirements.

7.2. Prevent Resources of Computing Nodes from an Overload

The fulfilment of this requirement is mostly dependent on the efficiency of the allocation algorithm, which is out of a scope of this paper. We have found at least one allocation algorithm that accomplishes the goals of the reconfiguration service. We used a simple allocation algorithm in order to prove the concept of the service. The algorithm chooses a random component from the most overloaded node and migrates it to the least loaded node. Due to migration of components, we achieved a decreasing total value of the system overload. Results from the experiments are presented at the figure 6 for different scenarios of components allocations. Each scenario denotes different components allocation on three computer nodes at the initial system state. In the scenario 1, 7 components, scenario 2- 12 components, and the scenario 3 – 17 components are allocated. Presented results are representative for proving the concept of the service. The data trend line indicates the decreasing value of the overload in the entire system. The more efficient allocation algorithm would improve these results. Moreover results are also dependent on the overload definition. In our exemplary system, we defined the overload as presented in the table 2.

Time [sec]	Processor idle [%]
0-5	0-10
0-15	10-25
Sustained	75-100

Table 2 Overload definitions

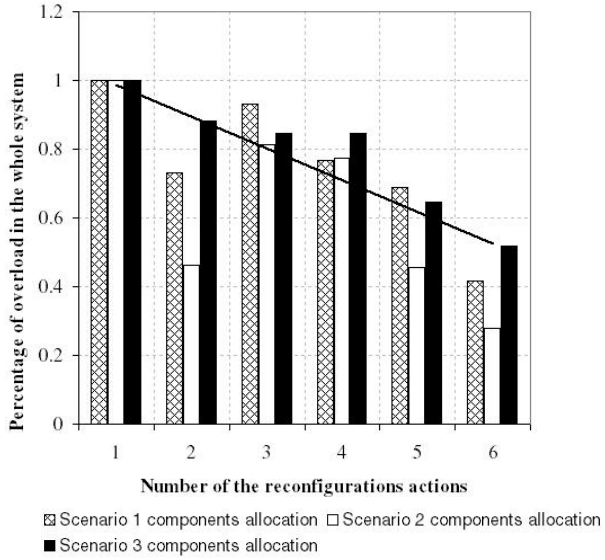

Fig. 6. Influence of the performance of reconfigurations actions for overload value.

7.3. Reliability of the Components

The Reconfiguration Manager keeps track of all running components and nodes in the system (the role of the centralized database). When it does not receive information about the components expenditure of resources, from Reconfiguration Agent, it raises the event of component failure. The instruction of the reconfiguration action is sent to Reconfiguration Agent to re-instantiate the component (see figure 7).

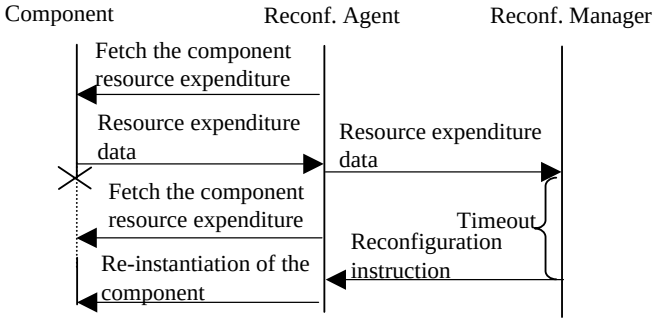

Fig. 7. Interaction diagram of the re-instantiation of the component

In the situation of the node failure, the Reconfiguration Manager does not receive any information from Reconfiguration Agents operating on the failure node. Then it sends instructions to other Reconfiguration Agents to re-instantiate the components, which were operating on the failure node.

The reconfiguration service does not operate at the network layer, rather then at application layer. Hence it cannot detect the network failure. Any case of the network failure is considered as the node failure. Another limitation is dealing with temporary failures e.g., a component or node appears to be crashed, at it is re-instantiated, and then old component re-appears.

These are the limitation of our solution, which we plan to eliminate in a future development of the service.

We use mean time to repair (MTTR)[3] as an evaluation criterion for a reliability. Through conducted measurements we found out that the mean time of detecting faulty components and restoring component to operation is 3,2 seconds (figure 8). This value is dependent on the frequency of the component monitoring, which we arbitrarily fixed at 1 Hz. The impact of the reconfiguration service with frequency of monitoring 1 Hz does not negatively influences the overall system performance (see section 7.5). We conclude that MTTR value of 3,2 seconds is acceptable for the domain of non-real time applications. There is a trade-off between MTTR time and the created overhead by the reconfiguration service.

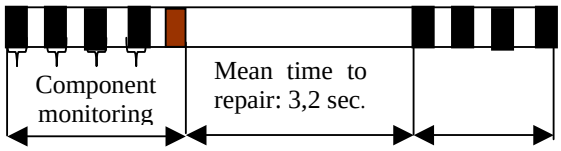

Fig. 8. Component monitoring

7.4. The Reconfiguration Service Overhead

The service overhead is defined in terms of resources expenditure of components that implement the service. We assumed that control messages introduce negligible network traffic. Figure 9 presents the average expenditure of resource by Reconfiguration Manager and three Reconfiguration Agents, also during the performance the reconfigurations actions.

Based on conducted measurements, we conclude that, the proposed reconfiguration service with monitoring frequency 1 Hz is relatively little intrusive process into the overall system performance.

[3] MTTR is the average amount of time needed to restore a faulty component to specified conditions.

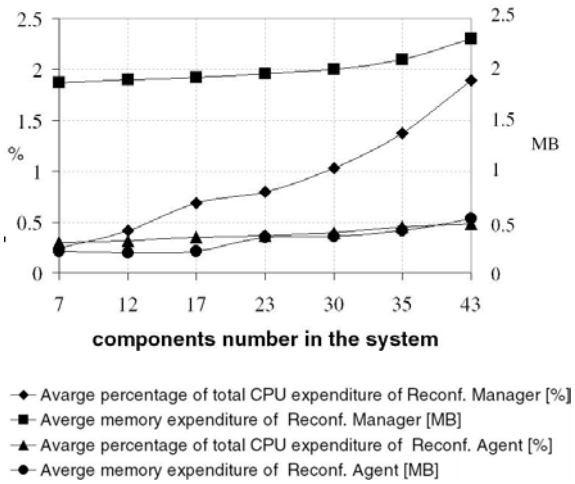

Fig. 9. Reconfiguration service overhead

8. Related Work

An alternative approach to ours for reconfiguration in p/s middleware is presented in [10.] The dynamic reconfiguration is defined informally as the ability to rearrange the routes traversed by events in response to changes in the topology of the network of dispatchers (components of architecture), and to do this without interrupting the normal system operation. This is contrary to our approach in which we assume changes in the components allocation in the fixed topology of the network.

The Lira infrastructure for managing dynamic recon• guration applies and extends the concepts of network management to component-based, distributed software systems [18]. Lira is designed to perform component-level reconfigurations through Reconfiguration Agents associated with individual components and the latter through a hierarchy of managers. Reconfiguration Agents are programmed on a component-by-component basis to respond to reconfiguration requests appropriate for that component. Managers embody the logic for monitoring the state of one or more components, and for determining when and how to execute re-configuration activities [18]. The taken approach seems to be similar to ours, however this work does not discuss anything about: the correctness preservation, state consistency during reconfiguration, the impact of reconfiguration infrastructure on the system performance, and components reliability.

9. Conclusions

The major contribution of this paper is the application of the dynamic reconfiguration concept (for example as presented in work [9]) to the publish/subscribe middleware. It has resulted in a design of the high-level architecture of the reconfiguration service. The service diminishes drawbacks of a static components allocation approach, and brings in significant benefits. These are the guarantees of predictable and reliable behaviour of the components in the context of mission-critical systems. This has been achieved due to preserving resources of computing nodes from an overload, monitoring components and accomplishing components deployments requirements.

Measurements show that our prototype implementation of the service has a low overhead.

The important contribution of the paper is the observation that the particularity of p/s communication model and a design choice of approach ensure correctness preservation, and state consistency during the reconfiguration.

The adopted data-centric, computational model (one-to-one mapping between a software component and an operating system process) significantly simplifies component migration, measurements of components expenditure of resources, and allows applying process placement algorithms to the middleware system.

10. Future work

An imperfection of components migration approach can result in lost of a small piece of data during a migration. In a future we plan to extend the reconfiguration service by "mechanism", which would enforce components to reach "safe state" in order to prevent lost of the data.

Another matter worthy of a consideration is duration of the synchronization of local caches, and how does it influence for duration of the reconfiguration.

We also plan to eliminate the limitations of our solution in a future development of the service.

Acknowledgements

This work was partly supported by European Research Programme: Marie Curie Host Fellowship under the contract number: HPMI-CT-2002-00221

This work was partly supported by the European Union under the E-Next Project FP6-506869

References

[1] G. Cugola, H.A Jacobsen - Using Publish/Subscribe Middleware for Mobile Systems -ACM SIGMOBILE Mobile Computing and Communications Review archive;Volume 6 , Issue 4 (October 2002); Pages: 25 - 33

[2] R. Joshi, G-P Castellote - A Comparison and Mapping of Data Distribution Service and High-Level Architecture, RTI whitepaper; http://www.rti.com

[4] Gero Muhl - Large-Scale Content-Based Publish/Subscribe Systems – Dissertation Vom Fachbereich Informatik der Technischen Universitat Darmstadt 2002;

[5] K. Raymond – "Reference Model of Open Distributed Processing (RM-ODP): Introduction", IFIP International Conference on Open Distributed Processing (ICODP'95), Brisbane, Australia, Febr. 1995

[6] Eberhardt Rechtin, Mark W. Maier - ,"The art. of systems architecting", 1997 by CRC Press, Inc, ISBN: 0-8493-7836-2

[7] L.P Peixoto Dos Santos "Application Level Tun Time Load Management A Bayesian Approach" – PhD Thesis, Universidade Do Minho 2001

[9] Maarten Wegdam – "Dynamic Reconfiguration and Load Distribution in Component Middleware" – PhD thesis, University of Twente, CTIT Ph.D-thesis series, No. 03-50, ISSN 1381-3617; No. 03-5, 26 June 2003.

[10] G. Cugola, G.P.Picco, A.L. Murphy- Towards Dynamic Reconfiguration of Distributed Publish-Subscribe Middleware. In Proceedings of the 3rd International Workshop on Software Engineering and Middleware (SEM02), co-located with the 24th International Conference on Software Engineering (ICSE02), May 2002, Orlando (FL), USA, A. Coen-Porisini and A. van Der Hoek eds., Lecture Notes on Computer Science vol. 2596, pp. 187-202, 2003.

[11] [DDS_SPEC] Data Distribution Service for Real-Time Systems Specification, ptc/03-07- 07

[12] K. Moazami-Goudarzi, "Consistnecy preserving dynamic reconfiguration of distributed systems", PhD thesis, Imperial College, London, March 1999

[13] L.P Peixoto Dos Santos "Application Level Tun Time Load Management A Bayesian Approach" – PhD Thesis, Universidade Do Minho 2001

[14] Schantz RE and Schmidt DC 2002 Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications In Encyclopedia of Software Engineering (ed. Marciniak J and Telecki G) Wiley & Sons New York.

[15] N. Wang, C.D. Gill, D.C Shmidt, A. Gokhale, B. Natarajan, J. P. Loyall, R.E. Schantz, C. Rodrigues – "QoS-enabled Middleware" from book Middleware for Communication. Edited by Qusay H. Mahmoud, @2001 J. Wiley & Sons, Ltd.

[16] CISCO Documentation http://www.cisco.com/univercd/cc/td/doc/cisintwk /ito_doc/snmp.htm

[17] Wanqian David Liu - A Distributed Data Flow Model For Composing Software Services - dissertation at Stanford University - June 2003

[18] M. Castaldi, A. Carzaniga, P. Inverardi, A.L. Wolf -A Lightweight Infrastructure for Reconfiguring Applications - B. Westfechtel, A. van der Hoek (Eds.): SCM 2001/2003, LNCS 2649, pp. 231–244, 2003. c Springer-Verlag Berlin Heidelberg 2003

[19]Th. Eugster Felber. The many faces of publish/subscribe. Technical report, Swiss Federal Institute of Technology in Lausanne (EPFL), 2001.

[20] G. Pardo-Castellote – OMG Data Distribution Service: Real-Time Publish/Subscribe Becomes a Standard- RTC Magazine, January 2005

[21] B. Zieba, M. Glandrup, M. van Sinderen, M. Wegdam- Reconfiguration Service for Publish/Subscribe Middleware Systems - submitted to the EuroPar '05