Position Paper on Dependability and Reconfigurability in Distributed Embedded Systems

Julián Proenza
SRV, Dept. de Matemàtiques i Informàtica
Universitat de les Illes Balears, Spain
julian.proenza@uib.es

Luís Almeida
IEETA, DETI
Universidade de Aveiro, Portugal
lda@det.ua.pt

Abstract

Dynamic Reconfiguration (DR) has been generating a substantial interest since it allows improving efficiency in the use of system resources, which can impact both on the maximum functionality that the system can execute, on the level of resources needed for a given functionality, on the number of instantaneous users that the system can support, or even on the capacity to adapt to changes in the environment or on the system operational architecture such as those caused by hazardous events. However, DR also requires extra mechanisms to manage the reconfiguration itself, which can increase system complexity and reduce a priori knowledge, increasing the potential for lower reliability. Therefore, DR has not been considered in safety-critical systems. In this paper we argue that adequately preventing specific error situations at the lower levels of the architecture simplifies the upper-level systemwide Fault Tolerance mechanisms, and may compensate for the extra complexity and lower a priori knowledge that DR implies, thus opening the door to the construction of highly-reliable dynamically reconfigurable systems.

1. Introduction

Reconfigurability has long been recognized as a way to improve efficiency in the use of system resources [22], for example, when a system undergoes variable load situations, when it evolves during its lifetime or even when faults affect part of its structure [25]. This means that reconfigurability, in a broad sense, may be beneficial to areas that range from Quality of Service (QoS), e.g., when the number of system users varies [20], to Dependability, e.g., through *graceful degradation* [26].

However, achieving reconfigurability may conflict with operational goals such as continued real-time and safe operation, and it becomes more difficult when the system is distributed, requiring adequate support from the network. Hence, whenever either of those two operational goals are relevant, the typical option has been to rely on a single static configuration [7][27]. In some cases, reconfigurable solutions have been devised but limited to few predefined operational modes, thus still with reduced flexibility and efficiency [27]. Conversely, for QoS purposes, reconfigurability seems to bring along clear benefits [20] and the conflicting goals referred for the case of safety critical systems do not seem to apply.

In this paper we discuss the interaction between Dynamic Reconfiguration (DR) and Fault Tolerance. We show that, despite its higher efficiency, DR introduces new dimensions in the system state space and, mainly, new mechanisms to mediate and enforce the system state changes. In other words, DR reduces the a priori knowledge concerning the exact state the system is in and introduces extra mechanisms that may lead to higher complexity and thus lower reliability. We, then, discuss ways to compensate for such negative aspects and propose using hardware-implemented mechanisms to prevent specific error situations at the lowest levels of the architecture, in order to simplify the upper-level systemwide Fault Tolerance mechanisms and improve their coverage. Several examples of recent related work will be referred. The remainder of the paper is organized as follows, Section 2 discusses the definition of DR, Section 3 discusses how DR is sometimes used to improve the system dependability making use of existing system resources, Section 4 discusses the limitations of DR in what concerns Fault Tolerance aspects and Section 5 presents the proposed solutions. Section 6 concludes the paper.

2. On the concept of DR

DR is a broad concept that spans many application domains. A common or integrated perspective of DR, including a taxonomy and boundaries of what is and what is not DR is still to be done despite recent initiatives in that direction [1]. Concerning this paper, it is important to separate the concepts of DR and Fault Tolerance. Notice that Fault

Tolerance mechanisms typically involve some kind of on-line reconfiguration, e.g., disconnecting nodes affected by faults and replacing them with spares or adding new nodes to compensate for disconnected replicas. In this sense, Fault Tolerance mechanisms are a subset of DR. However, these mechanisms normally aim at maintaining the same functionality, only, despite the occurrence of faults.

On the other hand, DR is normally taken in a broader way, considering changes in the allocation of tasks and messages to resources (e.g. nodes, links, bandwidth and energy), or even changes in the operational parameters of the system (e.g. scheduling and control parameters). The purpose of DR is typically to improve resources usage, considering the resources that are already available because they are needed for the basic functionality of the system. It is also common to consider that DR implies a high level of flexibility / adaptability in the system.

Therefore, Fault Tolerance mechanisms based on replication are not normally taken as DR. On the other hand, as discussed in the next Section, DR can still be used to improve the dependability of systems that were not designed as traditional fault-tolerant systems., e.g. by reallocating system resources that were currently available, possibly providing *graceful degradation*.

3. Improving dependability with DR

The idea of taking advantage of the already available resources relates DR to the low-cost Fault Tolerance approach of using *Unintentional Redundancy* [17]. This kind of redundancy is usually available in all systems (particularly in distributed ones). We illustrate next how Unintentional Redundancy with DR can be used to tolerate faults.

First, in some interconnection topologies, such as the one in Fig. 1, there are several paths that can be used to connect each pair of nodes. This opens doors for dynamically reconfigurable architectures that, in case of failure of one link, reestablish the communication using an alternate path. For example, in Fig. 1, in case the direct link between nodes 1 and 2 is faulty, the communication between these two nodes can be reestablished though nodes 3 and 4. A specific example of this kind of DR is described in [3].

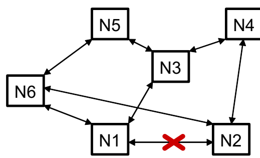

Figure 1. A network performing DR

Second, in general distributed systems, the presence of multiple nodes enables the reallocation of tasks from faulty nodes to non-faulty ones. For example, in Fig. 2, in case node 2 is known to be faulty its tasks could be assigned to another node. An example of such type of reconfiguration in an automotive system has been pointed out in [19]

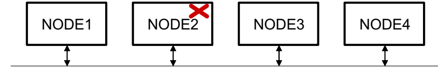

Figure 2. Dynamic reallocation

4. Limitations of DR

Obviously, the use of Unintentional Redundancy with DR has a limited capacity for Fault Tolerance. Among the reasons for this, we point out two. First, depending on the system, the available redundancy can be not enough to tolerate some faults. Therefore single points of failure may still exist. And second, in many cases the reconfiguration causes a *graceful degradation*, which means that either the level of performance has been decreased or even the system is no longer fully functional and some supposedly non-critical functions have been shutdown. Notice that graceful degradation is, normally, a positive feature in the sense that what would be a global failure is exchanged by an operating configuration that still provides a reduced level of QoS. Nevertheless, for certain critical systems that reduced level of QoS might not be sufficient to meet the minimum operational requirements.

Therefore, when using Unintentional Redundancy, DR is a suitable means to achieve a general improvement of the system dependability, but it might be not so well suited to reach the high levels of dependability that are usually pursued by fault-tolerant architectures. Moreover, dynamic reconfigurations are likely to take some time to complete, which, depending on the specific nature of the system, might also be constrained by the dynamics of the environment. This might rule out the use of DR in critical systems with fast dynamics unless special architectures that support fast reconfiguration are used, e.g. [12].

4.1. Addition of resources for DR

Despite the efficiency improvement in using already available resources being one of the typical characteristics of DR, the truth is that some resources must be added to a system for it to be able to perform DR. In particular, many systems that perform DR do it thanks to the inclusion of suitable mechanisms in their middleware. Moreover, some hardware additions can be also used, which, as will be discussed below, may provide an advantageous support for a safe reconfiguration.

However, too many additions would represent a deviation from the initial target of achieving low cost by effi-

ciently using the available resources. Therefore, a trade-off has to be found between cost and functionality.

4.2. DR means flexibility, complexity and overhead

As referred before, DR is supposed to imply a (high) flexibility in the system. But, flexibility also means complexity. In the case of a distributed system, this complexity appears in two forms. First, nodes have to perform new actions for the system to be able to react in the face of various situations and to adapt to the ever changing reality. As indicated above, these new actions are usually implemented in the middleware. And second, the communication channel has to transport an increased number of messages, e.g. to coordinate the reconfiguration among nodes.

Therefore, DR normally introduces a computational and communication overhead that may be not acceptable for many distributed embedded systems based on low-performance microcontrollers and low-bandwidth communication technologies. Moreover, beyond these overheads, an increased complexity usually means a decreased reliability.

4.3. Facets of unreliability in DR

The unreliability caused by the increased complexity of systems performing DR is essentially provoked by the increased number of scenarios that the system has to be designed to deal with, arising from the multiple possible configurations that the system can adopt and the faults that the system must react to.

The presence of these multiple scenarios makes it much more difficult to achieve the so-called *systemwide integration of fault tolerance*. This integration is pointed out in [2] as one of the fundamental steps in the design of complex fault-tolerant systems, since those are prone to suffer failures caused by improper interactions among their non-faulty subsystems. Some examples of parts that are difficult to integrate are the Fault Tolerance mechanisms that are intended to deal with the local faults of each subsystem with those that provide Fault Tolerance for functions that are executed as a global cooperation among several subsystems. This systemwide integration also has to prevent improper interference among concurrently active recovery or reconfiguration algorithms.

Similarly the increased number of scenarios to deal with also make it difficult the qualitative evaluation [2] of the system. This kind of evaluation is intended to verify that the design of the system includes all the mechanisms which are necessary to deal with the expected classes of faults. The higher the number of error scenarios is, the higher the difficulty in verifying the correct operation of the system in all these scenarios will be. More specifically, *model checking* [8] is likely to become a standard evaluation procedure for fault-tolerant systems in the next few years, much in the same way as simulation is already a de facto requirement in the development of computing systems. Although modern model checkers such as UPPAAL [18] already exhibit an enhanced capacity to deal with large state spaces, the nature of model checking makes this technique quite vulnerable to the complexity of the systems to be modeled and verified. The amount of memory required to generate the state space of complex models makes model checking useless in practice for the verification of highly-complex systems.

For all the reasons discussed above, keeping the complexity of a system performing DR under reasonable bounds should receive the maximum attention when dependability is the main concern.

5. Reconciling DR and Fault Tolerance

A way of reducing the complexity, and thereby increasing the reliability, is to reduce the number of scenarios that the system has to deal with, in particular those scenarios created by the faults that the system has to tolerate.

The techniques to be used in order to achieve this reduction of scenarios are implemented in the form of hardware additions that prevent specific error situations as close as possible to the faults that generate them. Since part of the errors that the subsystems can suffer are resolved as close as possible to their origin, the upper layers of the system architecture, such as the middleware, are much less complex and more reliable for they do not have to deal with the aforementioned error situations or only have to deal with simplified ones.

Examples of these techniques are, first, the use of specific circuitry to restrict the *failure semantics* [9] of the nodes. If Byzantine or arbitrary failures of the nodes are not possible, the software of the other nodes does not have to deal with them. And second, the use of hardware-implemented communication protocols that provide consistent communication services. Thereby the communication channel does not cause additional complex scenarios, e.g., inconsistencies in the delivery of messages, to be resolved by the upper layers of the architecture. For instance, the *MajorCAN* protocol [23] is a slightly modified version of the *Controller Area Network* (CAN) [14] that truly provides atomic broadcast at the data-link layer.

5.1. Defining error containment boundaries

An adequate restriction of the nodes failure semantics has the additional effect of preventing a node from acting as a *babbling idiot* [16] that, by sending messages at the wrong moment, blocks the communication channel and impedes the exchange of messages among nodes.

In more general terms, the restriction of the failure of semantics is a significant help to prevent the propagation of

errors from a faulty node to the rest of them. In the same manner, the use of hardware-implemented communication protocols that provide consistent communication services also helps to prevent that errors in the communication are propagated to the receiving nodes.

However it is important to note that the failure semantics restriction together with the use of consistent communication services is not enough in order to completely define an *error containment boundary* [2] around each node (Fig. 3) because, after all, even if we restrict the failure semantics of a node, it is possible for it to suffer a failure. In order to cope with these situations it is also very important to design the other nodes' software (e.g. middleware) in such a way that they are able to recognize these failures and to properly react to avoid their effects. This is much easier to achieve when the failure semantics is restricted, e.g. if nodes present crash failure semantics, node failures can be detected by timing out on regularly transmitted *I am alive* messages. Designing the global operation of the system to work properly in the event of node failures is an additional concern of the design of any truly fault-tolerant system.

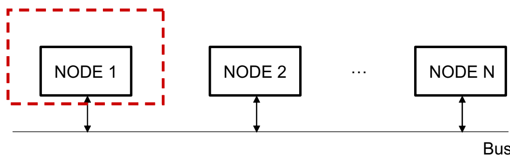

Figure 3. Error containment boundaries

5.2. Advantages of defining low-level error containment boundaries

Beyond the aspects that have been already discussed (e.g. reduction of the middleware complexity and thus increase of its reliability), using techniques at the lower levels of the system architecture to prevent error propagation exhibits a number of additional relevant advantages.

First, it reduces the overhead generated by the communication since no higher-layer protocols are required in order to ensure consistency. This has two facets, on the one hand less messages are transmitted and, on the other hand, less computation time is devoted in the nodes to the tasks related to communication.

Second, less nodes are required. Since failure semantics are restricted, the requirements on the number of nodes to be used in order to achieve agreement on a value under byzantine failure semantics [10] are relaxed.

And third, it prevents the so-called *amplification of failures* [13]. A typical high-level implemented consistent communication service requiring several rounds of message transmissions uses lower-level (and less dependable) communication primitives, such as point-to-point message *sends* and *receives* [13]. In this kind of high-level communication services, the broadcast of a message requires the execution of several instructions, and may include several sends and receives. It is well known that in this kind of complex communication schemes a failure at the low level of send and receive primitives (e.g. an omission to send a message) does not necessarily manifest at the high level as the same type of failure (e.g. an omission to broadcast a message to all receivers). In fact, it is said that this kind of broadcast algorithms are likely to amplify the importance of failures which occur at the low level [13] (e.g. messages delivered to different receivers in a non consistent order due to an omission to send a message). Therefore by substituting this kind of high-level implemented protocols by low-level services already presenting the required properties we would make it possible to eliminate this risk.

It is important to note that the set of advantages pointed out above are achieved when both failure semantics restriction and low-layer consistent communication services are used at the same time. However we do not claim that both features must always be included in the architecture. Even if only one of the features is used, significant reduction of the potential error scenarios is obtained.

5.3. Failure semantics restriction implementation

When using any mechanism to restrict the failure semantics of the nodes it is very important to achieve a high probability for the final node design to exhibit the pursued failure semantics, i.e., to have an *assumption coverage* [21] as high as possible. After all, the design of the rest of the distributed system is based on assuming said failure semantics.

To enforce a restriction in the failure semantics of the nodes several design techniques can be used. One of the most effective ones is *duplication with comparison* [15]. This technique is based on using two identical pieces of hardware actively performing the same operations in parallel, and comparing the results of said operations. In case there is a discrepancy in the results, an error signal is activated. This allows the detection of errors in the duplicated module and can be used to restrict the failure semantics by using the error signal to disconnect the node from the network (e.g. disabling the communication transceiver). This scheme is shown in Fig. 4. In this simple manner a *crash failure semantics* is enforced, meaning that the node either works properly or crashes. For a high assumption coverage to be achieved using this technique it is important to duplicate as much parts of the circuitry as possible. For example, [24] and in [12] describe CAN nodes with internal duplication and comparison that can disable their transceivers in case of discrepancy.

Another technique for restricting the failure semantics of a node in a bus-based architecture is the incorporation of a

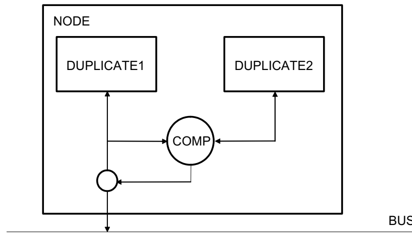

Figure 4. Duplication with comparison for failure semantics restriction

bus guardian [11]. This mechanism is devoted to prevent a node acting as a babbling idiot from keeping the bus busy with the transmission of useless messages, thereby making it more difficult for the other nodes to communicate. Several different bus guardians have been reported in the literature, either for time-triggered and event-triggered communication as well as for flexible communication requirements such as the one described in [12] for the FTT-CAN protocol.

5.4. Other error containment techniques

Having to modify the structure of each of the nodes is unacceptable in some applications due to cost reasons. In these cases a different approach for failure semantics restriction can be adopted. This new approach is based on the use of star topologies instead of buses. Such a topology allows to include mechanisms for error detection and *fault passivation* [17] (e.g. disconnecting the faulty nodes from the network) in the hub and, thereby, standard hardware can be used for the nodes. Fig. 5 illustrates this idea. By placing the error containment mechanisms into the hub and by preparing all nodes to deal with the scenarios caused by faulty nodes an equivalent definition of error containment boundaries can be achieved. Note that the hub may prevent the propagation of errors generated either in the nodes, in the links that connect each node with the hub or even in the circuits that implement the communication protocol.

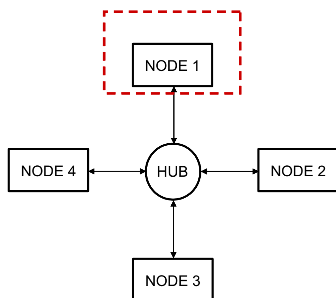

Figure 5. A star for error containment

Besides the capacity of an active hub to enforce a restricted failure semantics for each node of the network, the aforementioned capacity of preventing the propagation of the errors caused by faults in the circuits that implement the communication protocol (including cables) is a significant advantage of the star topology when it is compared to a bus [6]. The main drawback of the bus topology is that the structure of the network presents multiple components, which have direct electrical connections to each other without proper error containment. As a consequence, a fault in any of them may generate errors that propagate and effectively prevent further communication to take place.

In contrast, a star topology reduces this multiplicity of potentially failure-generating components to only one, the hub. Although it is clear that a star significantly reduces the probability of having more than one node affected by a failure (most of the errors caused by a single faulty component are not allowed to affect more than one node) [5], in some applications the presence of the single point of failure that the hub represents is unacceptable. In these cases redundant star topologies can be used [4].

5.5. Bringing it together

As referred before, we consider that it is possible to reconcile Dynamic Reconfiguration with high levels of Fault Tolerance as long as nodes failure semantics restriction and low-layer consistent communication services are incorporated into the system design from the beginning. Most of the actual components needed for the particular case of CAN technology have been developed by our groups along the past 8 years. Specifically, we consider that the FTT-CAN protocol is naturally adapted to support prompt reconfiguration under continued timeliness and that the mechanisms presented in [12] can be further simplified and the overall Fault Tolerance features improved, mainly concerning their analyzability, by merging that protocol with MajorCAN [23] to achieve true atomic broadcast, and the (Re)CANcentrate hubs [4] for strong error containment.

6. Conclusions

Dynamic Reconfiguration is gaining a growing interest as a way to improve the efficiency in using system resources. Moreover, DR has also been pointed out as a way to achieve inexpensive Fault Tolerance, e.g., by means of graceful degradation. However, combining DR with high levels of FT raises several problems, mostly related with the reduced a priori knowledge of DR systems and with the reconfiguration mechanisms themselves that introduce extra complexity and overhead, thus lower reliability. In this paper we have discussed the interaction between DR and FT and we have proposed combining nodes failure semantics restriction and low-layer consistent communication services to simplify the system middleware layers and improve

their analyzability. This will allow building highly reliable and resource efficient systems that are capable of adapting to the environment, to systems changes or to different load situations while tolerating the designated faults.

7. Acknowledgement

This work is partially supported by DPI 2005-09001-C03-02 and FEDER funding.

References

[1] Neres 2007 - artist2 workshop on networks for reconfigurable embedded systems. `http://www.artist-embedded.org/artist/-NERES-2007-.html`, April 2007.

[2] A. Avižienis. Building dependable systems: How to keep up with complexity. In *Special Issue of the IEEE 25th Int. Symp. Fault-Tolerant Computing. FTCS-25. Pasadena, CA*, pages 4–14, June 27–30 1995.

[3] D. Avresky and N. Natchev. Dynamic reconfiguration in computer clusters with irregular topologies in the presence of multiple node and link failures. *IEEE Transactions on Computers*, 54(5):603–615, May 2005.

[4] M. Barranco, L. Almeida, and J. Proenza. ReCANcentrate: A replicated star topology for CAN networks. In *Proceedings of the 2005 IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2005). Catania, Italy*, 2005.

[5] M. Barranco, J. Proenza, and L. Almeida. First results of the assessment of the improvement of error containment achieved by CANcentrate. In *Proceedings of the 6th IEEE International Workshop on Factory Communication Systems (WFCS 2006). Torino, Italy*, 2006.

[6] M. Barranco, J. Proenza, G. Rodríguez-Navas, and L. Almeida. An active star topology for improving fault confinement in CAN networks. *IEEE Transactions on Industrial Informatics*, 2(2):78–85, May 2006.

[7] Belschner, R. *et al*. FlexRay Requirements Specification, version 2.0.2. *FlexRay Consortium, `http://www.flexray-group.com`*, 2002.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[9] F. Cristian. Questions to ask when designing or attempting to understand a fault-tolerant distributed system. In *Keynote Address in Proc. 3rd Brazilian Conference on Fault-Tolerant Computing. Rio de Janeiro, Brazil*, September 1989.

[10] D. Dolev. The byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

[11] J. Ferreira, L. Almeida, and J. Fonseca. Bus guardians for can: a taxonomy and a comparative study. In *Proc. of WDAS 2005, Workshop on Dependable Automation Systems*. Brazilian Computing Society, October 2005.

[12] J. Ferreira, L. Almeida, J. Fonseca, P. Pedreiras, E. Martins, G. Rodriguez-Navas, J. Rigo, and J. Proenza. Combining operational flexibility and dependability in FTT-CAN. *IEEE Transactions on Industrial Informatics*, 2(2):95–102, May 2006.

[13] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. J. Mullender, editor, *Distributed Systems*, ACM-Press, chapter 5, pages 97–145. Addison-Wesley, second edition, 1993.

[14] ISO. *International Standard 11898 – Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High-Speed Communication*. 1993.

[15] B. W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems*. Addison-Wesley Publishing Company, 1989.

[16] H. Kopetz. A node as a unit of failure. In *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Real-Time Systems. Engineering and Computer Science, chapter 6.3, pages 129–131. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.

[17] J.-C. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag Wien New York, 1992.

[18] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[19] J. Li, Y. Song, and F. Simonot-Lion. Providing real-time applications with graceful degradation of qos and fault tolerance according to (m, k)-firm model. *IEEE Transactions on Industrial Informatics*, 2(2):112–119, May 2006.

[20] R. Moghal and M. Mian. Adaptive QoS-Based Resource Allocation in Distributed Multimedia Systems. In *Proceedings of Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, 2003.

[21] D. Powell. Failure mode assumptions and assumption coverage. In *Digest of Papers of the IEEE 22th Int. Symp. Fault-Tolerant Computing FTCS-22*, pages 386–395, Boston, Massachusetts-USA, July 1992.

[22] D. Prasad, A. Burns, and M. Atkins. The Valid Use of Utility in Adaptive Real-Time Systems. *Real-Time Systems*, 25(2-3):277–296, 2003.

[23] J. Proenza and J. Miro-Julia. MajorCAN: A modification to the Controller Area Network protocol to achieve Atomic Broadcast. In *Proceedings of the IEEE Int. Workshop on Group Communications and Computations. IWGCC. Taipei, Taiwan*, April 2000.

[24] J. Proenza, J. Pons, and J. Miro-Julia. A low-cost fail-safe circuit for fault-tolerant control systems. In *Proceedings of the 6th IEEE Int. Conf. on Electronics, Circuits and Systems . ICECS'99. Pafos, Cyprus*, September 1999.

[25] D. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma. Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems, CrossTalk, November. 2001. `http://www.cs.wustl.edu/~schmidt/PDF/crosstalk.pdf`; accessed February 21, 2005., 2001.

[26] C. Shelton and P. Koopman. Improving system dependability with functional alternatives. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 295. IEEE Computer Society, 2004.

[27] TTTech. Time-Triggered Protocol TTP/C High-Level Specification Document (edition 1.0). *`http://www.ttagroup.org`*, 2002.