Broker Fault Recovery for a Multiprocessor System-on-Chip Middleware

Anderson R. P. Domingues*, Jean Carlo Hamerski*†, Alexandre Amory*
*Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
†Instituto Federal de Educação Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Porto Alegre, Brazil
anderson.domingues@acad.pucrs.br, jean.hamerski@restinga.ifrs.edu.br, alexandre.amory@pucrs.br

***Abstract*—The publish-subscribe programming model has been used successfully in many distributed embedded application domains and has been recently ported to the MPSoC domain. However, the publish-subscribe model requires the element of the broker, which is a single process that manages the communication between nodes; a unique point of failure in the system. This paper presents a lightweight extension of the publish-subscribe model with a fault recovery method for the broker. The results show that the proposed method inserts small memory footprint to the system while providing minimal system downtime during recovery.**

***Index Terms*—fault tolerance, embedded software, parallel distributed applications, MPSoC, publish-subscribe.**

I. Introduction

Multiprocessor system-on-chips (MPSoCs) emerged as an attractive computer architecture for high performance embedded applications [1]. It usually consists of a set of small embedded processors, typically connected by a network-on-chip (NoC). As the number of available processors increases, it also increases the challenges to manage such systems efficiently. Several applications are competing for resources while some overall parameters such as application throughput, maximal power dissipation, and energy consumption have to be balanced at runtime. Moreover, as the technology node decreases in size, enabling more transistors per area, it also increases the probability of having faults occur in chips, whether these faults were generated during the manufacturing process or circuit aging caused them. These challenges become more problematic when considering that the chip might have some defectives parts. Thus, even though it is currently possible to manufacture MPSoCs with hundreds or thousands of processors, the ability to design self-managing applications, able to deal with several conflicting constraints and possible faults, is still a research problem [2].

Since the design of parallel distributed applications is already challenging, it is advisable to hide from the application level the additional issues related to self-managing systems by providing high-level functionalities and services for the programmer. Other distributed embedded application domains such as Internet of Things [3] and Robotics [4] typically provide a middleware, which is a layer of software that provides a set of services to the application level, hiding implementation details and abstracting the hardware platform. One of the preferable programming models on those distributed embedded application domains is called publish-subscribe (pub-sub) [5], which decouples the source and destination of the messages by using named data channels called *topics* and a process called *broker* responsible for connecting the channels to the interested processes.

One drawback of this programming model is that the broker is a single process that, if unavailable due to a fault, causes the entire systems to crash. Thus, the broker is a critical point of failure in the pub-sub programming model. Since MPSoCs typically consist of several small processors with a limited amount of memory (e.g., about hundreds of Kbytes), usual fault tolerance (FT) approaches such as TMR (triple module redundancy) and checkpoint-rollback are not recommended.

This paper presents a lightweight fault recovery mechanism for brokers of a publish-subscribe middleware for MPSoCs. The proposed approach uses the existing brokers to backup sensitive data of its neighbor brokers, which provides high availability to the system because when a fault is detected in a broker's processor, its neighbor broker promptly assumes the responsibility of managing the applications of the faulty broker. This broker replacement is entirely transparent to the application level. Note that the approach is loosely coupled to the hardware as it relies on few features, such as parallel architecture, distributed private memories, message passing API, and a clustered managing system.

This paper is organized as follows. Section II presents the related work with a focus on system-level and user-level FT for MPSoCs. Section III describes the original pub-sub middleware for MPSoCs. Section IV presents the proposed extensions for broker fault recovery. Section V presents the analytical models for network message volume estimation. Section VI shows the MPSoC platform case study and experimental results. Finally, Section VII concludes the paper.

II. Related Work

Recent works aim to provide FT on the user-level and system-level, specifically on hardware/software elements that have some management function in the MPSoC. For user-level, Barreto et al. [6] present a software-based method that automatically reallocates to a healthy processing element (PE) the tasks of an application affected by faults. The application is restarted in the new PE, causing a short downtime.

978-1-5386-7431-4/18/$31.00 © 2018 IEEE

Regarding system-level, Wachter et al. [7] propose a specialized control NoC, in addition to a primary data NoC, used to detect faults on PEs and to discover new fault-free paths on the NoC. The approach requires additional hardware elements to accomplish FT. Zou and Pasricha [8] propose a system-level framework to trade-off energy consumption and fault-tolerance in the NoC fabric. The framework addresses energy-efficient resilience in NoCs in two phases: design time and runtime. At design time, the authors implement an application mapping heuristic while satisfying application bandwidth and latency constraints. At runtime, a prediction algorithm dynamically estimates fault vulnerability of NoC router components, to manage energy overheads by enabling or disabling FT mechanisms in the NoC. Also focusing on the system-level, Fochi et al. [9] present a method to fault detection with a protocol to migrate the management software to another healthy PE. The goal is to preserve managed data without using redundant structures. The authors use an auxiliary NoC to detect the fault in a PE. A keep-alive protocol performs the fault detection. After the fault has been detected, a neighbor management PE copies the kernel and context memory to a slave PE chosen between those that compose the cluster. The method only covers CPU faults, requiring both the router, network interface, and memory to be fault-free.

The related works cited in this section present essential advances to improve fault detection and tolerance in MPSoC environments. Those works that focus on the system-level require for additional hardware [7–9] to accomplish the FT. The work presented in this paper focuses on the system-level, specifically in the broker management FT, with detection and actuation entirely performed in software components. In this paper, a fault is inserted at the broker's PE and, as in similar works, we also assume that the network and other PEs are fault free.

III. Publish-Subscribe System

The publish-subscribe pattern is an alternative to perform communication between participants (processes) of a parallel distributed system. In this communication model, the sender process (publisher) does not send messages directly to a specific receiver process (subscriber). The messages are classified into topics of interest, and a subscriber receives messages only of those topics to which it has subscribed. Publishers send messages to topics without knowledge of which are the subscribers since the pub-sub system coordinates the communication between publishers and subscribers at system-level. In most pub-sub systems, a component named broker is responsible for this coordination, either intermediating or coordinating the subscriptions.

This paper is based on the Message-Queuing SoC (MQSoC) pub-sub middleware for MPSoCs [10]. The operations supported by this middleware are: (i) *advertise*, which announces a new topic to the system; (ii) *unadvertise*, which makes a topic unavailable to the rest of the system; (iii) *subscribe*, with which a process announces that it wants to receive data from a specific topic; (iv) *unsubscribe*, where a process stops receiving data from a specific topic; (v) *publish*, whence a publisher sends data to a specific topic; and (vi) *yield*, which is used to schedule incoming messages in a process. In this middleware, the broker is involved in the four initial operations (i, ii, iii and iv). Thus, they must be extended to implement the proposed broker fault recovery process. All operations are described as follows, such as the modifications made to support our approach.

A. Advertise

When a process performs an advertise, a message of type *msg_advertise* is sent to the broker informing the *address* of the publisher process and the *topic name* where the messages will be published. The broker then stores this data into its *Publishers* table and forwards the generated entry to a secondary broker, which stores a copy of the primary brokers' *Publishers* and *Subscribers* table. The operation ends by the publisher receiving a *msg_advertise_ack* from the primary broker, as shown in Figure 1(a).

B. Unadvertise

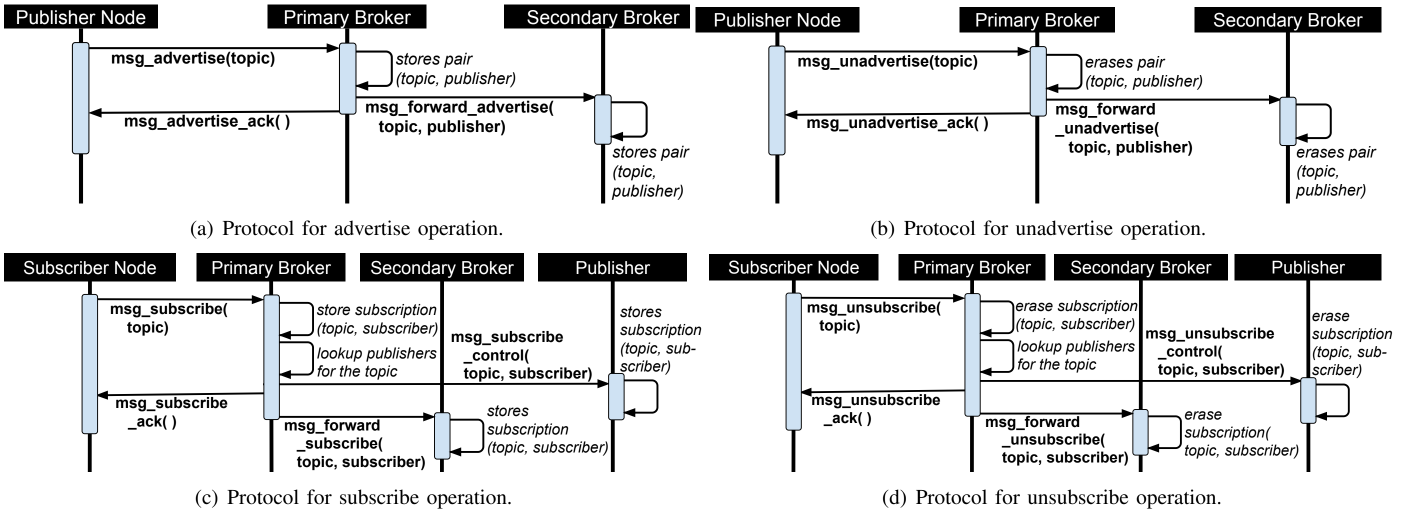
The unadvertise operation undoes a previous advertise operation. A message of type *msg_unadvertise* is sent to the broker which erases the *publisher address* and the informed *topic name* from its *Publishers* table if such an entry exists. As in the advertise operation, data is also forwarded to the secondary broker, which also erases its entry copy. At the end of the operation, the publisher receives a *msg_unadvertise_ack* informing that the operation succeeded, as shown in Figure 1(b). Note that publishers cannot send any more messages via this specific topic unless they advertise on the topic again.

C. Subscribe

A process can subscribe to a topic by sending a *msg_subscribe* message to the broker informing its *address* and the *topic name* of the subscription. The broker then stores the data in its *Subscribers* table and lookup for publishers for the topic into its *Publishers* table and forwards it to the secondary broker. Messages of type *msg_subscribe_control* are sent to each of the publishers informing that further messages must also be sent to the new subscriber. Finally, the subscriber receives a *msg_subscribe_ack* message to confirm the operation, as shown in Figure 1(c). It is important to note that the same topic might have multiple publishers and multiple subscribers at the same time.

D. Unsubscribe

A subscriber sends a *msg_unsubscribe* to the broker informing its *address* and the *topic name* to unsubscribe from. Then, the broker removes the respective entry from the *Subscribers* table and forwards the unsubscription to the secondary broker. The broker also searches for the publishers of this topic in its *Publishers* table and sends them *msg_unsubscribe_ctrl* so that they stop sending messages to the subscriber for that topic. The protocol is shown in Figure 1(d).



(a) Protocol for advertise operation.

(b) Protocol for unadvertise operation.

(c) Protocol for subscribe operation.

(d) Protocol for unsubscribe operation.

Figure 1. Protocol for each of the publish-subscribe middleware operations.

E. Publish

The publish operation allows publishers to feed a topic by sending messages of type *msg_publish* to the processes subscribed to that topic. Since publishers are aware at middleware-level of each of the subscribers for topics they advertised, the message is sent directly to each of the subscribers. Brokers do not take part in this operation, which is thus not affected by faults in the brokers.

F. Yield

The yield operation permits a subscriber to wait for a certain number of publications for some topic. This operation blocks the subscriber in a loop until messages are received. The buffer is periodically verified for messages and, in case of a message in the buffer, the associated callback is called and the message counter decreased by one. Otherwise, the task could be suspended until a message is received if the optional parameter *suspend* is given.

IV. Proposed Fault Tolerance Protocol

The proposed fault recovery approach relies on a 3-stage protocol consisting of monitoring, cluster recovery and broker recovery phases. Before presenting these stages, we have to present the general backup structure of the brokers. We assume that the entire system is divided into clusters of nodes where one node in the cluster is called cluster manager and has the role of broker for the middleware. These brokers are related as presented in Figure 3. The arrow in the figure indicates that the broker at the end of the arrow (called primary broker) is monitored by the broker at the start of the arrow (called secondary broker). This ring topology is reconfigured in the presence of a faulty broker.

A. Monitoring Phase

In the first phase, a secondary broker periodically sends messages of type *msg_keepalive_request* to its respective primary broker 2(a). The primary broker then must respond in time with a *msg_keepalive_response* or otherwise it will be considered as faulty (Figure 2(b)). The time between requests and responses must be configured according to system needs. A very long time may delay the detection of faults, whereas a very short time may overload system's network.

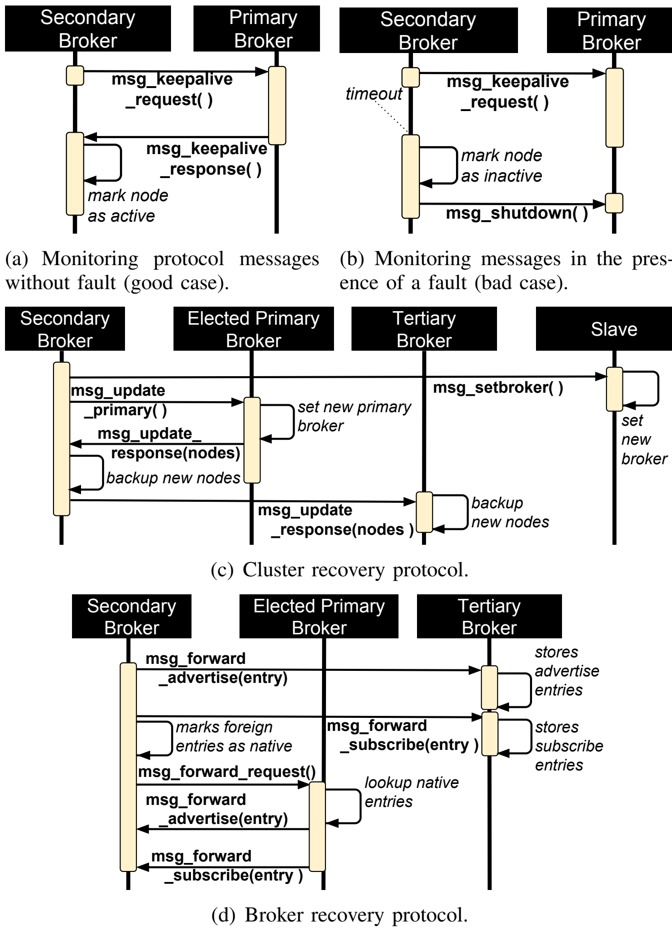
Once a secondary broker detects that its respective primary broker is faulty, the recovery protocol begins. The faulty node receives a message of type *msg_shutdown*, requesting its shutdown. This message is to ensure that secondary broker and the primary broker will not be serving the cluster at the same time, forcing the faulty broker out of the network.

As the last step, the secondary broker must inform other brokers that its primary broker is now out of the system. So, the secondary broker sends a *msg_fault_advertise* to the new elected primary broker, which sends the message to its secondary broker (which the former is tertiary to) and so on and so forth. The message keeps being sent until it reaches all the brokers in the system.

B. Cluster Recovery Phase

This phase can be seen in Figure 2(c). Secondary brokers keep a list of members of its cluster and another list of members of its primary broker cluster. These lists are generated at brokers' startup and are not modified unless some fault is detected. The cluster recovery starts by merging both lists. From this point, all slaves inside the cluster in which the fault was detected will be part of the cluster of the secondary broker. The secondary broker sends a *msg_set_broker* message to each of the affected slave nodes.

The recovery proceeds with the secondary broker updating its own secondary broker (i.e., the one which monitors it) by sending the address of new members in the cluster. Also, the cluster members of the elected primary broker must be requested, so the secondary broker can back them up and recover in case of another fault.



(a) Monitoring protocol messages without fault (good case).

(b) Monitoring messages in the presence of a fault (bad case).

(c) Cluster recovery protocol.

(d) Broker recovery protocol.

Figure 2. Protocols for each of the phases of the proposed FT mechanism.

C. Broker Recovery Phase

Brokers store a list of pairs of subscribers and subscribed topics and another list of publishers and topics they advertise. Every time an advertise, unadvertise, subscribe or unsubscribe operation takes place, these lists are updated, and copies of the updated entries are sent to the secondary broker. It is achieved during each of the middleware's operations through the messages *msg_forward_advertise*, *msg_forward_subscribe*, *msg_forward_unadvertise* and *msg_forward_unsubscribe*.

The broker recovery starts with the secondary broker concatenating its lists with the ones of the faulty broker. Since these lists represent topics, both clusters were merged into one. The secondary broker is now the primary broker of the merged cluster. A new primary broker is elected to be monitored, replacing the faulty broker. The ring configuration is then repaired and, finally, entries for Publishers and Subscribers tables of the elected primary broker are backed up. The broker recovery phase is shown in Figure 2(d).

V. Analytical Models

Enabling the FT module implies additional messages being sent through the network. The number of messages might vary from system to system and is bound to implementation. In this section, we introduce analytical models (AM) for estimating the communication overhead caused by our proposed protocol, assuming that every message has the same length.

A. Monitoring Phase

As presented in Section IV-A, *msg_keepalive_request* are periodically sent by secondary brokers to keep up to date with a primary broker's status. These messages can be answered with a *msg_keepalive_response* or be missed. In both cases, the number of messages stays the same for a given time span, since a miss will generate a *msg_keepalive_shutdown*. Considering l as the length of the time span between two *msg_keepalive_request* and t the point of time when a fault is detected, the number of messages (Φ) sent is given as shown by Equation 1.

$$\Phi = 2t.l^{-1} \tag{1}$$

B. Cluster Recovery Phase

During cluster recovery, slaves of the faulty broker receive *msg_set_broker* messages informing that their broker changed to the secondary broker (M_f). The tertiary broker must also be informed of new members, generating an additional message per cluster member. Elected primary brokers update the secondary broker with its entries (M_n) in response to the *msg_update_request*. Finally, the *msg_fault_advertise* is propagated to all brokers once per fault (B). The total of messages for cluster recovering phase (Ψ) is shown in Equation 2.

$$\Psi = 2M_f + M_n + B + 1 \tag{2}$$

C. Broker Recovery Phase

Brokers exchange *Publishers* and *Subscribers* tables entries during the broker recovery phase. The secondary broker elects a new primary broker and sends a message of type *msg_forward_request* to it, receiving its entries in response for both Publishers and Subscribers tables ($P_n + S_n + 1$). Before storing the entries, the secondary broker must update the tertiary broker with its entries ($P_u + S_u$). The Equation 3 models the total of messages exchanged during the broker recovery phase (Υ).

$$\Upsilon = P_n + S_n + P_u + S_u + 1 \tag{3}$$

The total volume of messages Π for all phases of the protocol is given in by the sum of Φ, Ψ and Υ, which comes from Equations 1, 2 and 3. It is important to note that these equations do not relate to each other and can be used independently to estimate message volume for each phase of the protocol. For instance, we could find the total volume of messages generated during cluster recovery for the application presented in Section VI-B, as demonstrated in equation 4.

$$\begin{aligned} \Psi &= 2M_f + M_n + B + 1 \\ &= 2 \times 8 + (8 + 4 + 1) = 29 \text{ messages} \end{aligned} \tag{4}$$

VI. Experimental Results

This section presents the MPSoC Platform (VI-A), the application adopted in the case study (VI-B), the FT setup (VI-C), and the respective experimental results (VI-D).

A. *FreeRTOS MPSoC Platform*

Figure 3 illustrates the case study platform composed by a 6x6 NoC-based MPSoC platform, with homogeneous processing elements (PEs) in 3x3 clusters. This configuration is used in all experiments. Each PE includes a Cortex-M4F processor, private random access memory (RAM) addressed to store both system and applications, network interface, DMA, and router.

The system running in each PE is composed of an extended FreeRTOS kernel [11] and the incorporated pub-sub middleware [10]. The kernel is responsible for the task mapping feature, as well as the task scheduling, and interface with the hardware elements. The applications are stored in a repository accessed only by PE-0 and are mapped at system startup during task mapping. The middleware is responsible for the publish-subscribe system management extended by the proposed broker recovery protocol.

The MPSoC hardware infrastructure was described using OVPSIM APIs by Imperas, which provides an instruction accurate simulation framework. The kernel software was implemented in the C programming language and the middleware software using the C++ programming language (compilers *arm-none-eabi-gcc* and *arm-none-eabi-g++*, version 4.9.3).

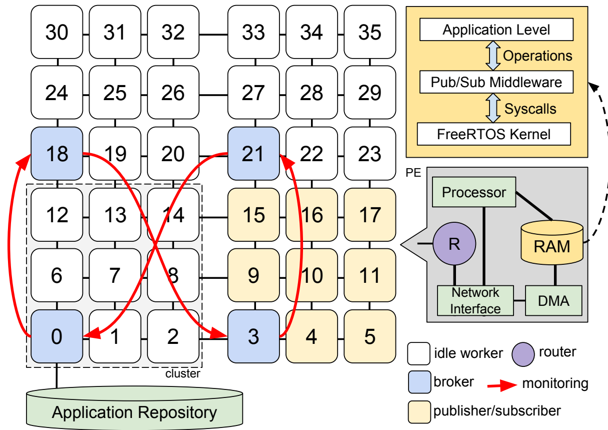


Figure 3. A 6x6 platform with 3x3 clusters configuration.

B. *Application Case Study*

The experiments use the Producer-Consumer application, which contains two tasks: a producer and a consumer. The producer task generates a message flow (workload) that is received by the consumer task. The pub-sub middleware operations are used to perform the communication between the two tasks. The Figure 4 shows the modeled producer-consumer application. A topic identifies the message flow. The producer task is the publisher, and the consumer task is the subscriber of the topic.

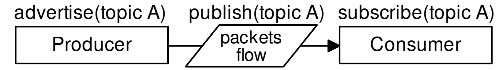


Figure 4. The producer-consumer application.

The only situation in which the proposed broker recovery protocol can delay an application is when the application requests actions for the broker while it is executing the recovery process. Thus, we devised a scenario to distribute in time the application requests for the broker, maximizing the probability of impact in the application execution time. The Figure 5 shows the scheme of performed test cases. The base scenario consists of four parallel producer-consumer applications (A, B, C, and D), which are spaced in the scenario execution time. Applications are configured with a workload of 32 messages. Application A is the first one to begin the execution ((1) in the Figure 5). In its turn, application B starts (2) when application A has performed half of its workload (16 messages). Applications C and D follow the same behavior. The scenario finishes when the application D workload is completed (3).

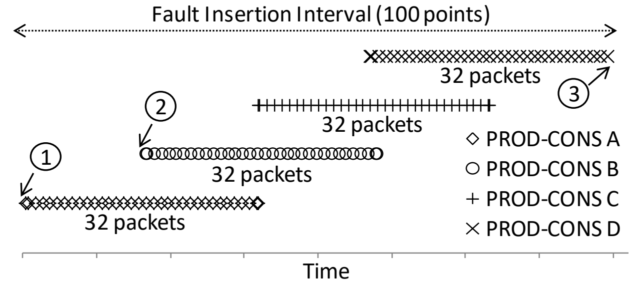


Figure 5. Application Case Study Scenario

C. *Fault Tolerance Feature Setup*

The broker FT feature has the following configurations that must be defined at design-time: (a) the time span between keepalive requests (30,000 clock cycles in the performed experiments); (b) the number of unanswered keepalive requests to consider the broker as faulty (3 in performed experiments).

D. *Results and Discussion*

The performed experiments use the scenario explained in the previous sections (VI-A and VI-B). In the 100 performed test cases, we evaluate the scenario execution time (SET), broker recovery time (BRT), and Communication Volume. The difference between the test cases is the moment when the fault is inserted into the broker (in 100 different instants of time). All test cases have the same task mapping scheme. Thus, they are mapped in the same cluster (colored in the Figure 3). The fault is inserted into PE-3, which is the broker manager of this cluster. The time unit is presented in thousands of clock cycles, measured by a timing model [11] that captures the executed instructions for each processor, generating an execution time in clock cycles from the total of executed instructions. The

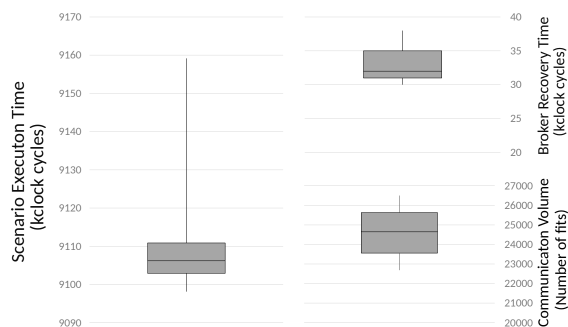


Figure 6. Box Plot Chart of the Scenario Execution Time (left), Broker Recovery Time (right-top), and Communication Volume (right-bottom) results for a sample of 100 test cases.

communication volume unit is flit. Messages are sent through the NoC divided by 32-bit wide flits. In this experimental setup, a control message has 16 flits.

Figure 6 shows the results of scenario execution time (SET), broker recovery time (BRT) and communication volume. The mean SET is 9,106 thousand of clock cycles. It presents a variation because the publish-subscribe communication suffers interference only at the test cases where a producer or consumer tasks call the advertise or subscribe operations at the same time as the broker recovery process occurs. The mean BRT is 32 thousands of clock cycles. The broker recovery presents an insignificant execution time compared with the SET, even if we consider that the evaluated application is very simple. When the BRT is compared to more complex applications, the impact is even smaller. The mean communication volume is 24,667 flits. This value represents flits of both control and data messages. This communication volume is much higher when compared to the analytical models in Equation 4. For comparison, we evaluate a scenario without fault insertion in the same experimental setup. The communication volume in this scenario is, on average, 5.04% higher because the four brokers remain exchanging keep-alive request/response messages in all scenario execution time. However, this network overhead can be trade-off with the recovery latency by changing the time span between keepalive messages.

Concerning memory footprint for code, the software system composed by the kernel and middleware with the proposed FT feature has 28.98KB, which corresponds to a 1.66KB increase compared to the middleware without the FT feature.

VII. Conclusion and Future Work

In this paper, we present a mechanism for FT in a communication middleware for MPSoCs. To achieve that we modified an existing middleware to provide data redundancy and extended the communication protocol to recover both brokers and cluster managers. We validated our approach through an experiment over a MPSoC platform. Results showed that our approach had a minimal resource overhead for different fault insertion setups. We also presented analytical models for estimating network usage.

Although our approach presented itself satisfactorily regarding consumed system resources, there are problems to be addressed regarding FT in brokers for the pub-sub model. For instance, there is no known solution for the treatment of transient faults for this domain. In this paper, we avoid the issue by disabling the faulty PE, even though it still has some operational capability. In the future, we intend to tackle this problem by extending the presented approach.

Acknowledgment

Thanks to Imperas Software Ltda. and Open Virtual Platforms for support and access to their models and simulator. Jean Carlo Hamerski is supported by CNPq and IFRS.

References

[1] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (mpsoc) technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, 2008.

[2] V. Nollet *et al.*, "A safari through the mpsoc run-time management jungle," *Journal of Signal Processing Systems*, vol. 60, no. 2, pp. 251–268, 2010.

[3] O. B. Sezer, E. Dogdu, and A. M. Ozbayoglu, "Context-aware computing, learning, and big data in internet of things: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 1–27, 2018.

[4] X. Li *et al.*, "A survey on intermediation architectures for underwater robotics," *Sensors*, vol. 16, no. 2, p. 190, 2016.

[5] P. T. Eugster *et al.*, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[6] F. F. Barreto, A. M. Amory, and F. G. Moraes, "Fault recovery protocol for distributed memory mpsocs," in *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 421–424.

[7] E. Wachter *et al.*, "Brnoc: A broadcast noc for control messages in many-core systems," *Microelectronics Journal*, vol. 68, pp. 69–77, 2017.

[8] Y. Zou and S. Pasricha, "Heft: A hybrid system-level framework for enabling energy-efficient fault-tolerance in noc based mpsocs," in *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2014, pp. 1–10.

[9] V. Fochi, L. L. Caimi, M. Ruaro, E. Wächter, and F. G. Moraes, "System management recovery protocol for mpsocs," in *System-on-Chip Conference (SOCC), 2017 30th IEEE International*. IEEE, 2017, pp. 367–374.

[10] J. C. Hamerski, G. Abich, R. Reis, L. Ost, and A. Amory, "Publish-subscribe programming for a NoC-based multiprocessor system-on-chip," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[11] G. Abich *et al.*, "Extending FreeRTOS to support dynamic and distributed mapping in multiprocessor systems," in *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2016, pp. 712–715.