A Fault Tolerance Extension to the Embedded CORBA for the CAN Bus Systems

Gwangil Jeon[1], Tae-Hyung Kim[2], Seongsoo Hong[3], and Sunil Kim[4]

[1] Seoul National University, School of Computer Science and Engineering, Seoul 151-742, Korea
gijeon@ssrnet.snu.ac.kr
[2] Hanyang University, Department of Computer Science and Engineering, Kyunggi-Do 425-791, Korea
tkim@cse.hanyang.ac.kr
[3] Seoul National University, School of Electrical Engineering, Seoul 151-742, Korea
sshong@redwood.snu.ac.kr
[4] Hongik University, Department of Computer Engineering, Seoul 121-791, Korea
sikim@cs.hongik.ac.kr

Abstract. This paper presents a fault tolerant extension to our CAN-CORBA design. The CAN-CORBA is an environment specific CORBA we designed for distributed embedded control systems built on the CAN bus. We extend it for fault tolerance by adopting passive and active replication strategies mandated by the OMG fault tolerant CORBA draft standard. To reduce resource demands of these fault tolerance features, we adopt a state-less passive replication policy and show that it is sufficient for embedded real-time control applications. We give an example CORBA program with its IDL definition to demonstrate the utility of our fault tolerant CAN-CORBA. The newly extended CAN-CORBA clearly reveals that it is feasible to use fault tolerant CORBA in developing distributed embedded systems on real-time networks with severe resource limitations.

1 Introduction

Many embedded system applications require a high degree of fault tolerance for their responsive and continuous operations since they often run in a harsh environment and possess stringent timing constraints. Unfortunately, writing programs for embedded systems, even without fault tolerant features, is already a seriously complicated task. Topping off fault tolerance requirements easily lead embedded system developers to the infamous embedded software crisis.

Recently, in [9] and [10], we proposed a new embedded CORBA design dedicated for CAN-based distributed control systems as an effort to provide a solution to the complexity problem of embedded software systems. We named this new CORBA design *CAN-CORBA* and demonstrated that it would be feasible to

use it in developing distributed embedded systems on real-time networks. There were two major difficulties we faced during the design of CAN-CORBA. First, the resource demands of the original CORBA implementation easily exceeded the 1 Mbps network bandwidth of the CAN bus. This required many optimizations in the transport protocol and inter-ORB protocol design for the new CORBA. Second, there was a crucial distinction in communication models between the conventional CORBA and typical control systems: connection-oriented vs. group communications. Our CAN-CORBA was designed to solve both problems.

In this paper, we extend our original design of CAN-CORBA for fault tolerance since it is one of inevitable requirements imposed on mission critical embedded real-time systems. Recently, the OMG (Object Management Group) made a request for a proposal to define a specification of fault tolerant CORBA [4]. At the time of submitting this paper, it was in the process of voting to adopt a standard fault tolerant CORBA after it had received the joint revised submission in December 1999 [6]. This submission was finally approved by the board of directors in March 2000.

As mandated by the draft specification, our fault tolerant schemes are based on object replication. For our CAN-CORBA, we adopt two replication strategies mandated by the draft fault tolerant CORBA specification. These are passive replication and active replication. Since the straightforward application of these strategies leads to excessive resource demands in embedded real-time systems, we propose a passive replication policy that does not require message logging and object state transfers. We show that such a state-less passive replication strategy is sufficient for applications running on top of our CAN-CORBA. We also show that active replication will be implemented in a straightforward manner in our CAN-CORBA due to the reliable broadcast bus of the CAN. To demonstrate the utility of our fault tolerance schemes, we give an example CORBA program with an IDL definition.

The remainder of the paper is organized as follows. Section 2 gives the target system hardware model and the publisher/subscriber communication schemes that our replication mechanisms are based on. In Section 3 we introduce the general replication strategies for fault tolerance and then present what are peculiar to the CAN-based applications and the CAN-CORBA transport protocols. Based on the identified peculiarities, we present the various replication mechanisms under the conjoiner-based CAN-CORBA transport protocols in Section 4. Section 5 presents an example program that demonstrates the usage of our extended transport protocol that includes fault tolerance in CAN-CORBA. Finally, Section 6 concludes this paper.

1.1 Related Work

Two issues are essential in designing a fault tolerant system: fault detection and replication management. Rajkumar and Gagliardi proposed a publisher/subscriber model for distributed real-time systems in [13] and developed a fault-tolerant extension to their publisher/subscriber model in [12]. Specifically, they adopted Cristian's periodic broadcast membership protocol [2] for fault detection. They

assumed that the underlying network was built on top of a point-to-point transport layer. As a result, one-to-many group communication was realized by sending message copies to multiple subscribers via a number of point-to-point connections. This assumption may lead to a performance problem since it is inefficient to simply send periodic "check" messages to detect failures – especially where many publishers and subscribers exist. Fetzer [3] devised an efficient mechanism for fail awareness under the publisher/subscriber communication paradigm. Since CAN provides a reliable broadcast medium, fault detection is hardly an issue in our work. It can be achieved through a simple timeout mechanism.

Synchronizing object states among replicas is one of difficult tasks in implementing a fault tolerant CORBA. Obviously, it is a computationally expensive operation and seriously complicates the resultant system. Unfortunately, the OMG fault tolerant CORBA RFP [4] mandates such a strong consistency and the joint revised submission [6] to the RFP proposes an ORB-level solution by notions of ReplicationManager, PropertyManager, and ObjectGroupManager, to name a few. On the other hand, the minimumCORBA [5] was proposed to remove dynamic facilities for serving requests and for creating, activating, passivating and interrogating objects. In embedded system design practice, decisions on object creation and resource allocation are usually made at design time. Moreover, as will be explicated in Section 3.2, and claimed by [12], the CAN-based embedded application does not require strong consistency among object replicas. Thus, we also excludes unncessary state management and dynamic object management features.

2 System Model

CAN-CORBA is designed to operate on a distributed embedded system built on the CAN bus. In this section, we present the target system hardware model and CAN-CORBA configuration to help readers understand the characteristics and limitations of the underlying system platform, before delving into the details of fault tolerance in our new CAN-CORBA design.

2.1 Target System Hardware Model

We use the same target hardware model as in [9] and [10] since we extend our prior implementation of CAN-CORBA to include fault tolerance. The target hardware consists of a number of function control units (FCU) interconnected by embedded control networks (ECN). As an example of the model, Figure 1 shows the electronic control system of a passenger vehicle. Each FCU, possessing one or more microcontrollers and microprocessors, conducts a dedicated control mission by interfacing sensors and actuators and executing prescribed control algorithms. Depending on configuration, an FCU works as a data producer, a consumer, or both.

As shown in Figure 1, embedded control networks (ECN) connect FCUs through inexpensive bus adaptors. Such ECNs are often required to provide

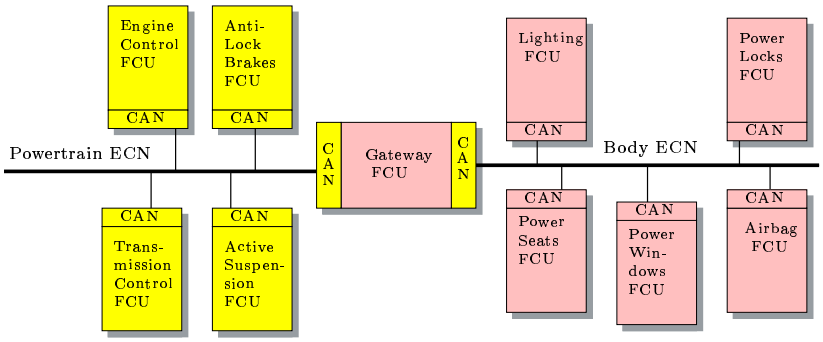


Fig. 1. Example distributed embedded control system: Passenger vehicle control system.

real-time message delivery services and subject to very stringent operational and functional constraints. In this work, we have chosen the CAN [7] as our embedded control network substrate since it is an internationally accepted industrial standard satisfying such constraints.

The CAN standard specifies physical and data link layer protocols in the OSI reference model [1]. It is well suited for real-time communication since it is capable of bounding message transfer latencies via predictable, priority-based bus arbitration. A CAN message is composed of identifier, data, error, acknowledgment, and CRC fields. The identifier field consists of 11 bits in CAN 2.0A or 29 bits in 2.0B and the data field can grow up to eight bytes. When a CAN network adaptor transmits a message, it first transmits the identifier followed by the data. The identifier of a message serves as a priority, and a higher priority message always beats a lower priority one.

The CAN provides a unique addressing scheme, known as subject-based addressing [11]. In the CAN, a message put into the network does not contain its destination address. Instead, it contains a subject tag – a predefined bit pattern in the message identifier which serves as a hint about its data content. A receiver node can program its CAN bus adaptor to accept only a specific subset of messages that carry a specific identifier pattern with them. This filtering mechanism is made possible via a mask register and a set of comparison registers on a CAN interface chip. This subject-based addressing scheme is a key underlying mechanism for the communication models of our CAN-CORBA.

2.2 CAN-CORBA Communication Channels

CAN-CORBA offers a subscription-based, anonymous group communication scheme that is often referred to as "blindcast" or as a publisher/subscriber scheme [13], [8]. In this scheme, a communication session starts when a data producer announces a predefined invocation channel. An invocation channel is a virtual broadcast channel from publishers to a group of subscribers. Data

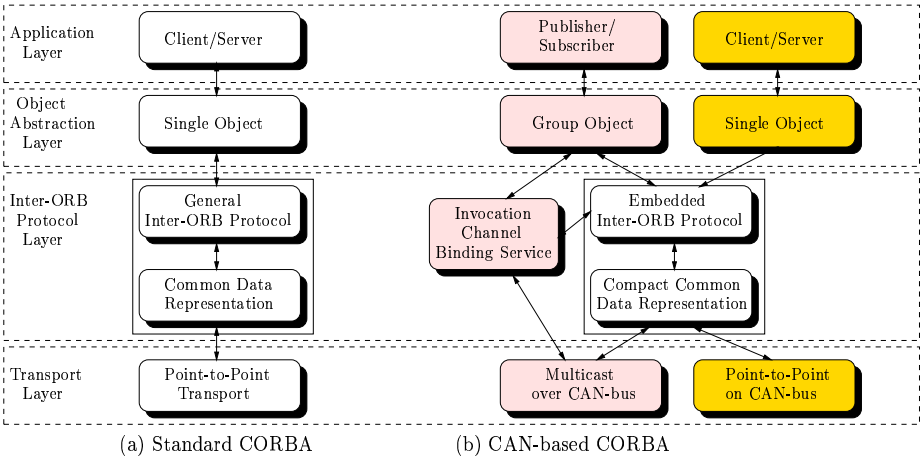


Fig. 2. Comparison between two CORBA configurations.

consumers can subscribe to an announced invocation channel. In this announcement/subscription process, neither a publisher nor a subscriber has to know each other. This anonymity allows for easy reconfiguration of control systems. In CAN-CORBA, an invocation channel is uniquely identified with a CAN identifier, and maintained by the *conjoiner* as described in Section 2.4.

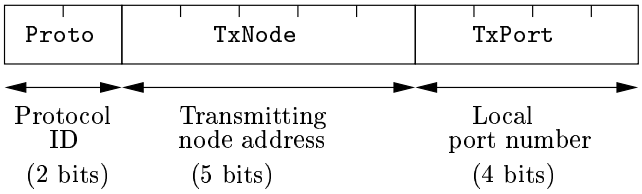
CAN-CORBA also provides point-to-point communication primitives for interoperability with other standard CORBA implementations. Since the CAN bus is a broadcast medium, the publisher/subscriber model is more natural and efficient than the point-to-point model. Moreover, fault tolerance is better serviced by a group communication scheme. Readers are referred to our previous work in [10] for more details on the connection oriented communications in CAN-CORBA.

2.3 CAN-CORBA Configuration

The proposed CAN-based CORBA design stems from the standard CORBA and possesses most of essential components of it. Figure 2 illustrates layer-to-layer comparison between the standard CORBA and the proposed one. Specifically, Figure 2 (b) shows our CAN-CORBA design.

We summarize the noticeable features of our CAN-CORBA.

- **Group object reference:** An object reference in CORBA refers to a single object. It is internally translated into an interoperable object reference (IOR) denoting a communication end-point the object resides on. In CAN-CORBA, an object reference may refer to a group of receiver objects. An intermediary object named a conjoiner is responsible for managing object groups and implementing the internal representation of their references.



Protocol ID	*Usage*
00	reserved
01	EIOP publisher/subscriber
10	EIOP point-to-point
11	binding

Fig. 3. Protocol header format using CAN identifier structure.

- **CAN-based transport protocol:** A new transport protocol is designed to support group communication in CORBA. In this protocol, a sender is totally unaware of its receivers and simply sends out messages via its own communication port.

- **Publisher/subscriber scheme:** A new communication scheme for the publisher/consumer model is also designed on top of the transport protocol. This scheme relies on an abstraction named an invocation channel. It denotes a virtual communication channel which connects a group of communication ports and a group of receivers. Since each port is owned by a publisher, this scheme supports the one-way, many-to-many communication model. In this scheme, a conjoiner object takes care of group management, dynamic channel binding, and address translation. An invocation channel is uniquely identified as a channel tag in an IDL program.

- **Compact common data representation (CCDR):** Common data representation is a syntax which specifies how IDL data types are represented in CORBA messages. In CDR, method invocations often take up tens of bytes in messages. Since a CAN message has only an eight-byte payload, a method invocation may well trigger a large number of CAN message transfers. To deal with this problem, we define the compact CDR. It exploits packed data encoding which avoids byte padding for data alignment, and introduces new data types for variable length integers to encode four-byte integers in a dense form.

- **Embedded inter-ORB protocol (EIOP):** In addition to CCDR, we customize GIOP by simplifying messages types and reducing the size of the IOP headers of messages.

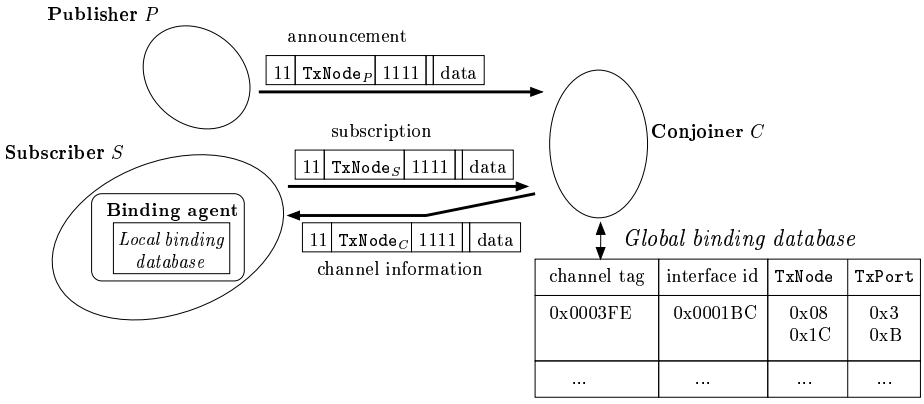


Fig. 4. Conjoiner-based channel binding protocol.

2.4 Channel Binding Protocol for Subscription-based Communication

Figure 3 shows the protocol header format. We divide the CAN identifier structure into three sub-fields: a protocol ID (`Proto`), a transmitting node address (`TxNode`), and a port number (`TxPort`). They respectively occupy two, five and four bits amounting to 11. The `Proto` field denotes an upper layer protocol identifier. The data field following the identifier in a CAN message is formatted according to the upper layer protocol identifier denoted by `Proto`. In the CAN, a message identifier with a smaller value gets a higher priority during bus arbitration.

The `TxNode` field is the address of the transmitting node. In our design, one can simultaneously connect up to 32 distinguishable nodes with the CAN bus under a given upper layer protocol. The `TxPort` field represents a port number which is local to a particular transmitting node. Since `TxNode` serves as a domain name which is globally identifiable all across the network, `TxNode` and `TxPort` collectively make a global port identifier. This allows ports in distinct nodes to have the same port number and helps increase modularity in software design and maintenance. As the `TxPort` field supports the maximum of 16 local ports on each node, up to 512 global ports coexist in the network under a specific upper layer protocol.

Our channel binding protocol relies on an intermediary object we name a *conjoiner*. It resides on a CAN node whose node identifier is known in advance to every publisher and every subscriber in the system. It must be started right after network initialization and operational during the entire system service period.

The conjoiner maintains a global binding database where each invocation channel has a corresponding entry which is announced and registered by a publisher. Figure 4 illustrates the conjoiner-based publisher/subscriber framework and the global binding database. As shown in the figure, an entry in the global

binding database is a quadruple consisting of a *channel tag*, an *OMG IDL identifier*, `TxNode` and `TxPort`. The channel tag is a unique symbolic name associated with each invocation channel. It is statically defined by programmers when they write the application code. Both publishers and subscribers use it as a search key in the global binding database later on. The OMG IDL interface identifier is a unique identifier associated with each IDL interface in the system. The OMG IDL compiler generates IDL interface identifiers. The CORBA run-time system uses these identifiers to perform type checking upon every method invocation. This ensures strong type safety as required by the CORBA standard. The channel tag and the interface ID together work as a unique name for each invocation channel. It is programmers' responsibility to define a system-wide unique name for an invocation channel.

The conjoiner exchanges messages with a publisher for channel establishment and with a subscriber for channel subscription. When a publisher wants to get attached to an invocation channel, it sends a registration message to the conjoiner. Similarly, a subscriber sends a message to the conjoiner, requesting subscription to an invocation channel. If the conjoiner finds an matching entry for the requested invocation channel in the global binding database, it provides the subscriber with the corresponding binding information. Note that subscribers may be asynchronously informed of changes in its subscribed invocation channel as a publisher is attached to, or detached from the channel. A local binding agent denoted by an oval inside `Subscriber` S node in Figure 4 takes care of such updates.

Note the conjoiner should be able to accept messages from any CAN nodes in the system. Thus, we reserve the local port number (`TxPort`) 1111_2 for this purpose under the network management protocol. As shown in Figure 4, all messages sent to the conjoiner use this local port. Consequently, the conjoiner unconditionally accepts all messages with this port number when `Proto` is 11_2. On the other hand, other CAN nodes can accept messages from the conjoiner in a straightforward way since the `TxNode` and `TxPort` of the conjoiner is known *a priori*.

The data field of a binding message carries full binding information or an actual query. Specifically, a publisher's registration message contains all necessary information to construct a database entry such as a channel tag, an OMG IDL interface ID, and a global port number. Using this information, the conjoiner either creates an entry or modifies one if it exists. A subscriber's request message contains a channel tag and an IDL interface identifier for an invocation channel. On successful retrieval of one or more entries from the binding database, the conjoiner sends a reply message containing information on these entries. These entries are stored into the local binding database of the subscriber.

3 Replication Strategies for Fault Tolerance

In general, fault tolerance is achieved through replication and the basic unit of replication is an individual object. Since non-fault tolerant application is re-

garded as a set of objects posessing single replicas in the program, the definition of fault tolerance may include that of non-fault tolerance, and it has an important subtlety since objects are interoperable with each other whether or not they are replicated.

We begin by introducing two general replication strategies that are mandated by the OMG fault tolerant CORBA draft standard [4], and explicate why stateless passive replication is sufficient for CAN-based applications.

3.1 General Replication Strategies

Replication is made to ensure continuous operation of a particular object. Depending on backing-up styles at the time of fault, replication strategy is classified as *passive* and *active* replication.

In passive replication, only one replica for a given object, which is called a primary replica, executes designated operations, and all others merely wait for an activating signal to be delivered when a fault is detected. According to the creation time of replicas and the length of a message log, it is further classified as cold and warm passive replication.

In *cold* passive replication, non-primary replicas are not created during normal operation, and method invocations and responses are recorded in a message log. When a fault is detected, a recovery service object is initiated and performs a given recovery action using the recorded message log. While this policy imposes no additional memory overhead as long as everything goes well, it requires long recovery time if a fault really occurs.

To shorten the recovery time of cold passive replication, all replicas can be created before faults occur, and the current state of a primary replica is periodically transferred to the others. The state transfer is made through multicast which must be reliable (i.e. not allowed to be lost) and totally ordered with respect to the time of state changes. In the event of a fault, one of the non-primary replicas can be substituted for the faulty primary by recovering only the state since it has been recently transferred. Thus, the recovery time can be reduced. It is called *warm* passive replication.

Note that the length of a message log determines the temperature of passive replication. If it is infinite, a message log should be recorded indefinitely until a fault occurs, thus no need to even create the non-primary replicas until a fault is detected. This is the cold end in the spectrum. If the length is zero, it means that no state is recorded for recovery but immediately transferred to the non-primary replicas via reliable and totally ordered multicast protocol. This is the hot end in the spectrum.

In active replication, when an object invokes a replicated service, all replicas service the request and actively reply with their own results without concerning the faults of other replicas. A client object collects results from all the replicas before making a final decision. Then it can choose one with majority voting, or without majority voting possibly based on the precedence among replicas.

3.2 State-less Replication Mechanism In CAN-CORBA

In our fault tolerant CAN-CORBA, the state of a primary replica need not be preserved or transferred to non-primary replicas. This allows us to replace the failed primary replica by one of non-primary replicas without transfering the state of the failed primary replica. This argument can be justified in the context of control systems theory. In general, control performance is seriously affected by the freshness of sampled data. Thus, when an embedded control system detects a run-time fault, it is more desirable for the system to start a new sampling period and produce actuation commands using recent data than to attempt to resume the interrupted sampling period using the restored state. As an example, consider a vehicle control system. If an object in an engine control unit does not receive oxygen-level data priodically published by an oxygen sensor, it may use data it received in the previous period. However, if the object cannot retrieve the previous state, possibly because it has been restarted due to a fault, it can re-establish the state using the data that are constantly provided from various publishers. As a matter of fact, it is recovered within a couple of minutes. Fuel efficiency may be lowered temporarily only for the adjusting period.

As a result, it is unnecessary to differentiate cold and warm passive replication for the fault tolerant CAN-CORBA. We thus employ a *state-less* passive replication policy. It is not hot passive since there is no state transfer between replicas during execution. It is not cold passive, either since a faulty primary is not substituted with a stand-by during recovery.

In the next section, we will present the replication strategies that are meaningful to our CAN-based CORBA system, and show the design of the fault tolerant CAN-CORBA using the conjoiner-based publisher/subscriber communication protocol.

4 Replicating CAN-CORBA Objects

As CAN-CORBA adopts the publisher/subscriber communication model for distributed inter process communication, we have three different entities for replication: publishers, subscribers and a conjoiner. Among them, publishers and subscribers are general CORBA objects and the conjoiner is a pseudo CORBA object that was deliberately invented to realize the publisher/subscriber communication model. We present how the replication strategies discussed in the previous section are applied to CORBA objects in our CAN-CORBA. We also show how to distribute and replicate a conjoiner in order to eliminate the single point of failure induced by the conjoiner.

4.1 Passive Replication

Figure 5 illustrates two configurations of a publisher/subscriber connection, each of which respectively denotes a situation before a fault and after its recovery using our state-less passive replication. In the figure, there is one publisher P and

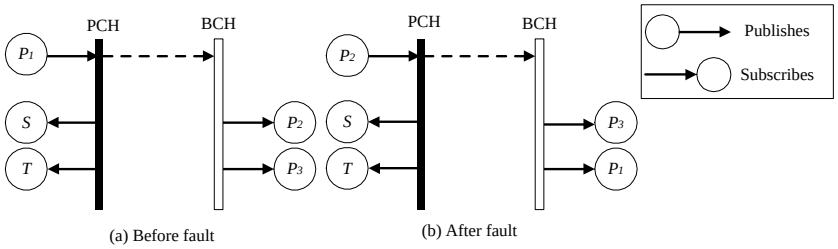


Fig. 5. Passive replication of publisher objects.

two subscribers S and T and their replicas are denoted by numbered subscripts. In our fault tolerant CAN-CORBA design, a publisher/subscriber connection is composed of dual channels: *primary* and *backup* channels. They are denoted by PCH and BCH, respectively. A primary channel (PCH) is used for normal communications while a backup channel (BCH) is used by non-primary publishers to monitor the state of the primary.

The publisher P has three replicas P_1, P_2, and P_3 for the invocation channel PCH where P_1 is the primary replica at the moment. In passive replication, faults are detected using timeouts. The primary replica broadcasts sensor data via PCH and heartbeat messages via BCH. Non-primary replicas do not publish anything and only monitor heartbeat messages. Thus, P_2 and P_3 are attached to BCH as the subscribers of the heartbeat messages. When a non-primary replica detects a fault through timeout, it requests the conjoiner to switch the channels.

The invocation channels after channel switching are shown in Figure 5 (b). Failed publisher P_1 is switched to backup channel BCH as a subscriber to P_2. P_2 becomes a new primary by attaching itself to both PCH and BCH as a publisher. Thus, all entries of {TxNode(P_1), TxPort(P_1)} must be replaced with {TxNode(P_2), TxPort(P_2)} in the global binding table. Becoming a non-primary replica is an autonomous operation performed by a failed primary replica. If P_1's fault is so fatal that it cannot update its local binding database, then it is eliminated from the application.

The following steps illustrate a scenario when P_1 misses the time to publish a heartbeat message.

(1) P_1 announces its registration to PCH and BCH.

(2) S and T request subscription to PCH.

(3) P_2 and P_3 request subscription to BCH.

(4) P_1 periodically publishes messages, S and T keep listening to PCH, and P_2 and P_3 keep monitoring P_1.

(5) P_2 (or P_3 or both) detects a timeout and requests channel switching to the conjoiner.

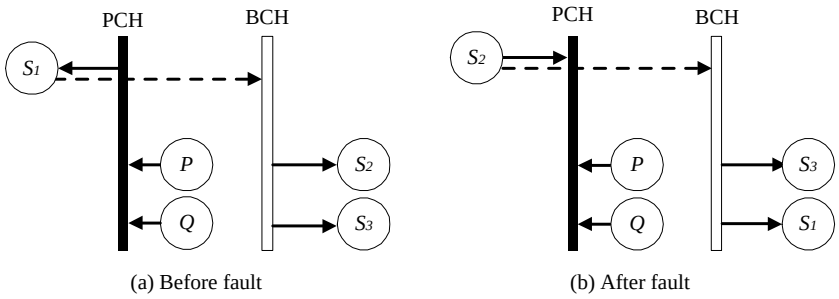


Fig. 6. Passive replication for subscriber objects.

(6) The conjoiner decides the next primary. It broadcasts the newly modified binding information which is a quadruple of (PCH, *primary interface*, TxNode(P_2), TxPort(P_2)). Upon receiving it, objects pertaining to the invocation channel performs the following tasks, respectively.
   - Subscribers S and T update their local binding databases. They have an entry of (PCH, *primary interface*, TxNode(P_2), TxPort(P_2)).
   - P_1 and P_2 switch the roles between primary and non-primary replicas. P_1 updates its local binding database by a quadruple of (BCH, *backup interface*, TxNode(P_2), TxPort(P_2)). If P_1 dies completely, it is discarded. P_1 will not be revived at all.
   - P_3 updates its local binding database. The resulting entry is (BCH, *backup interface*, TxNode(P_2), and TxPort(P_2).

(7) Step (6) completes channel switching. P_2 becomes a new primary and now starts to publish.

As a final note, there are several possibilities for the conjoiner to select the next primary among non-primary replicas. First, the conjoiner chooses in an FCFS fashion. When a fault is detected, many non-primary replicas may send publication requests to the conjoiner. The conjoiner knows the replicated set of publishers *a priori*. It takes the message that comes first and ignores all others. Second, the order is specified by an application programmer, for example, in a circular fashion using replica identifers. More generally, a selection function may be given in an IDL definition by a programmer.

Subscribers are replicated in a similar manner except that the primary subscriber periodically emits an "I am alive" signal to its fellow non-primary replicas. A primary publisher need not send this extra signal since messages periodically published can be used for this purpose. Figure 6 illustrates subscriber replication. The primary subscriber plays a role of a publisher to the non-primary subscribers. When the primary subscriber S_1 is failed and S_2 is substituted for S_1, the whole process is presented as follows.

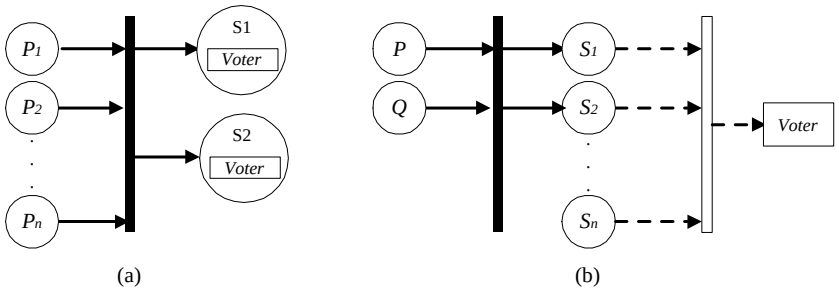


Fig. 7. Active replication for (a) publishers and (b) subscribers.

(1) S_1 requests its subscription to BCH.

(2) P and Q announces their registration to PCH.

(3) S_2 and S_3 request subscription to BCH.

(4) S_1 keeps subscribing to P and Q, sending "I am alive" message via *BCH*, and S_2 and S_3 keep monitoring S_1.

(5) S_2 (or S_3 or both) detects timeout and requests channel switching to the conjoiner.

(6) The conjoiner decides who is a new primary. The conjoiner broadcasts the newly modified binding information which is a quadruple of (BCH, *backup interface*, TxNode(S_2), TxPort(S_2)). The entry of (BCH, *backup interface*, TxNode(S_1), TxPort(S_1)) is eliminated because S_1 is no longer a publisher of "I am alive" message. Upon receiving it, objects pertaining to the invocation channel performs the following tasks respectively.
   - Publishers P and Q are not necessary to be aware of the change.
   - S_1 and S_2 switch the roles. S_1 and S_2 must update their local binding databases by sending queries with {BCH, *primary interface*}, {PCH, *primary interface*}, respectively, in order to switch the channels.
   - S_3 updates its local binding database with an entry of (*BCH*, *backup interface*, TxNode(S_2), TxPort(S_2)).

(7) Step (6) completes channel switching. S_2 is substituted for S_1 and now starts to subscribe.

4.2 Active Replication

In active replication for publishers, a subscriber must subscribe to one and each of replicated publishers, as illustrated in Figure 7 (a) where S and T subscribe to $P_1, \cdots, P_n$. Subscribers have a responsibility to multiplex all data from replicated publishers. A subscriber has freedom to adopt a multiplexing policy. It may use majority voting to tolerate commission faults that are semantically incorrect

data sent by publishers. Or it may serve data on an FCFS basis. In either case, a pertinent voting logic should be included in subscribers.

For subscribers that are actively replicated, an external voter object must be created. Each replicated subscriber publishes its decision to the voter, and the voter finalizes the decision. This is illustrated in Figure 7 (b). N-version programming can be implemented in this way.

4.3 Mixed Replication

It is natural to have mixed replication in an application. For example, it is possible that an actively replicated subscriber listens to a passively replicated publisher.

In our model, "publishers" act as data producers and "subscribers" as data consumers and an object can be a publisher and a consumer at the same time. When an object is replicated, this is done through manipulating invocation channels rather than modifying an entire object module. In a CAN-CORBA application, there are specific code segments that handle invocation channels. Structural changes are made on those code segments to specify replication strategies as will be illustrated in Section 5. Due to the regularity of replicated code structures, an automatic source-level translation scheme can be employed.

4.4 Replicated and Distributed Conjoiner

The conjoiner is an important system resource that takes care of channel binding and services channel switching requests. It also maintains a global binding database. In our original CAN-CORBA design, there is only one instance of a conjoiner, which poses serious reliability and performance problems.

To eliminate the single point of failure introduced by a centralized single conjoiner, the binding database is replicated. As a result, each binding entry is stored at more than two distinct locations. The conjoiner is actively replicated so that any of the replicated conjoiners can deliver correct binding information to its clients. Data consistency among replicated global binding databases is easily maintained using the reliable broadcast CAN bus.

To mitigate performance degradation due to a large number of binding and switching channel requests, the global binding database is distributed or fragmented. To efficiently service publisher announcement requests, each conjoiner replica inserts the binding entry into its database fragment only in its own turn. In this way, the number of entries among distributed conjoiners is balanced. When a subscription request is made, conjoiner replicas need to search only global binding database fragments and thus shorten the response time.

Recall that the presence of the conjoiner is known to all CAN nodes by unique `TxNode` and `TxPort` identifiers. Having multiple conjoiners, each conjoiner replica should have a unique `TxNode` assignment. On the other hand, any two or more conjoiner replicas that share the same disjoint subset of the global binding database should share the `TxPort` as well.

Suppose there are two disjoint sets of the binding database with four replicated conjoiners. The database has two fragments, `db1`, and `db2`. The two pairs of conjoiners (C_1, C_2) and (C_3, C_4) have `db1` and `db2`, respectively. We reserve two sets of local port number 1111_2 and 1110_2 in this case. Roughly speaking, the performance is doubled, and the probability of the conjoiner failure falls fifty percent.

5 Example Programs

```
// IDL
...
interface TemperatureMonitor {
  // Update temperature value for a location.
  oneway void update_temperature(in char locationID, in long temperature);
}

interface FaultMonitor {
  // Publish "I am alive" message.
  oneway void I_am_alive(void);
}
```

Fig. 8. IDL definition for non-primary publisher and subscriber interface.

In this section, we present an example program which demonstrates the usage of our transport protocol extended for the fault tolerant CAN-CORBA. It consists of an IDL interface definition (given in Figure 8), passively replicated publisher code (in Figure 10), and normal subscriber code (in Figure 9). This program denotes the case illustrated in Figure 5. Other combinations of replication such as passively replicated subscribers, and actively replicated publisher/subscriber pairs are similarly written.

The IDL code defines the interfaces of two invocation channels: primary channel (PCH) and backup channel (BCH). `TemperatureMonitor` and `FaultMonitor` interfaces contain the signatures of two methods `update_temperature()` and `I_am_alive()`, respectively. The `update_temperature` method is invoked by a publisher and then executed in a subscriber to update temperature within the subscriber object. It is declared as a `oneway` operation which does not produce output values. Obviously, two-way operation is not allowed in the publisher/subscriber communication protocol.

A primary publisher invokes the `I_am_alive()` method to notify its aliveness to its fellow passive publisher replicas that watch its fault by checking a timeout. Note that programmers may use a periodic message as a heartbeat message to reduce network traffic. In this particular example, an explicit heartbeat message is used, instead since this is more illustrative. Figure 9 and Figure 10 show two

source code files that correspond to a publisher and a subscriber, respectively. Each of the files contains unique channel tag TEMP_MONITOR_TAG and an IDL interface identifier TEMP_MONITOR_IFACE. Note that TEMP_MONITOR_TAG is defined by programmers while TEMP_MONITOR_IFACE is generated by our OMG IDL precompiler. Note that _B_TEMP_MONITOR_TAG and FAULT_MONITOR_IFACE are used only in publisher replicas.

```
// Define a channel tag for temperature monitoring.
#define TEMP_MONITOR_TAG 0x01

// Initialize the object request broker (ORB).
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv);

// Get a reference to the conjoiner.
Conjoiner_ptr conjoiner =
   Conjoiner::_narrow(orb->resolve_initial_references("Conjoiner"));

// Create a servant implementing a temperature monitor object.
TemperatureMonitor_impl monitor_servant;

// Assign a local CORBA object name to the monitor object.
PortableServer::ObjectId_ptr oid =
    PortableServer::string_to_ObjectId("Monitor1");

// Register the object name and servant to a portable object adaptor (POA).
poa->activate_object_with_id(oid, &monitor_servant);

// Bind the monitor object to the TEMP_MONITOR_TAG.
conjoiner->subscribe(TEMP_MONITOR_TAG, &monitor_servant);

// Enter the main to receive the temperature values.
orb->run();
```

Fig. 9. Subscriber code (not replicated).

In Figure 9, a subscriber wishing to subscribe to an invocation channel accesses the conjoiner object via `Conjoiner:: subscribe()` method. During this call, the subscriber provides the conjoiner with TEMP_MONITOR_TAG and a servant. A servant is a collection of language-specific data and procedures which implement the actual object body. It is written by an application programmer and registered into the CORBA object system via a portable object adaptor (POA). Note that the TEMP_MONITOR_IFACE is not explicitly provided during a call to `Conjoiner::subscribe()` method since the `monitor_servant` is a typed object whose interface information can be easily extracted. `Conjoiner::subscribe()` method sends a subscription request message to the conjoiner to get the binding information of an invocation channel. Finally, the subscriber enters into a block-

ing loop where it waits for an invocation of `update_temperature()` method from the publisher.

Figure 10 shows passively replicated publisher code that works for both primary and non-primary publishers. While a primary publisher works only as a data producer, non-primary replicas are in essence heartbeat data consumers that listen to a backup channel. Thus, the given publisher code includes invocations of both channel announcement and subscription. Programmers let the primary publisher register its primary and backup channels via two successive invocations of `Conjoiner::announce()` method. This method allocates a local port from the primary publisher's free port pool and sends an announcement message to the conjoiner. Finally, the primary invokes `update_temperature()` method to broadcast a temperature data message.

Programmers let the non-primary replicas subscribe to the backup channel via the invocation of `Conjoiner::subscribe()` method. During this call, the method provides the conjoiner with the backup channel tag and an object body `ft_detector` that implements the timeout checking logic. Non-primary replicas monitor the primary's status by periodically receiving "I am alive" messages. They wait inside a loop until `ft_monitor->I_am_alive()` is invoked by the primary or until a timeout occurs. If a timeout is detected, they request channel switching to the conjoiner by invoking `Conjoiner::switch()` method. This method returns true if the requesting replica is selected as a new primary. The conjoiner updates the global binding database to replace an old channel binding with a new one. As shown in Figure 3, `TxNode` and `TxPort` of the new channel binding are provided via the protocol header of our transport protocol. Finally, the program resumes publishing temperature data using a different sensor. When the primary channel is switched to the backup channel as a result of the recovery action, some of replicas experience timeouts. In that case, the local binding agent detects the channel switching by examining the local binding database.

6 Conclusions

We have presented a fault tolerant extension to our CAN-CORBA design [9], [10]. The CAN-CORBA is an environment specific CORBA we designed for distributed embedded systems built on the CAN bus. It supports both anonymous publisher/subscriber and point-to-point communications without losing the IDL level compliance to the OMG standard.

In order to support fault tolerance for CAN-CORBA applications, we adopted the OMG fault tolerant CORBA specification and incorporated into the CAN-CORBA both passive and active replication strategies. Since fault tolerance features added excessive complexity to the CAN-CORBA, we took into account the domain characteristics of the CAN-CORBA environment to avoid it. Specifically, we introduced a state-less passive replication policy that did not require message logging and object state transfers. We showed that state-less replication would be sufficient for embedded real-time control applications, as argued

```
// Define a primary channel tag for temperature monitoring.
#define TEMP_MONITOR_TAG 0x01

// Define a backup channel tag for passive replication.
#define _B_TEMP_MONITOR_TAG 0x101

// Initialize the object request broker (ORB).
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv);

// Get a reference to the conjoiner.
Conjoiner_ptr conjoiner =
    Conjoiner::_narrow(orb->resolve_initial_references("Conjoiner"));

// Obtain references to the temperature (PCH) and
// the replica (BCH) monitor groups.
TemperatureMonitor_ptr monitor =
   conjoiner->announce(TEMP_MONITOR_TAG, TEMP_MONITOR_IFACE);
TemperatureMonitor_ptr ft_monitor =
   conjoiner->announce(_B_TEMP_MONITOR_TAG, FAULT_MONITOR_IFACE);

// Create a servant implementing a fault detector object.
FaultMonitor_impl ft_detector;

// Assign a local CORBA object name to ft_detector.
PortableServer::ObjectId_ptr oid =
   PortableServer::string_to_ObjectId("Ft_detector1");

// Register the object name and ft_detector to a POA.
poa->activate_object_with_id(oid, &ft_detector);

// Bind the ft_detector object to the _B_TEMP_MONITOR_TAG.
conjoiner->subscribe(_B_TEMP_MONITOR_TAG, &ft_detector);

while(1) {

  // Each replica checks if it is a primary one. The conjoiner determines
  // the primary replica by returning TRUE to the ::switch() method.
  if (!conjoiner->switch(TEMP_MONITOR_TAG, TEMP_MONITOR_IFACE)) {

     // This is a main loop for non-primary replicas where the ft_detector
     // watches a time-out. This is terminated only if a fault is found.
     orb->run();

     // The ft_detector detects a fault. It is now terminated and requests
     // a channel switching to the conjoiner via invoking ::switch() method.
     continue;
  }

  // Primary replica starts here. Publish periodically.
  while(1) {

      // Invoke a method of subscribers.
      monitor->update_temperature('A', value);

      // Publish data to let replicas know my aliveness.
      ft_monitor->I_am_alive();

      if (conjoiner->is_switched()) break;
  }
}
```

Fig. 10. Passively replicated publisher code.

in [12]. Such a replication policy significantly helped reduce the complexity of our fault tolerant CAN-CORBA design. Since the fault tolerant CAN-CORBA provides replications for publishers, subscribers, and a conjoiner, not only can programmers be free from the single point of failure caused by the centralized conjoiner, but also they can freely add fault tolerance to their designs through replicating CAN-CORBA objects. Finally, we showed a program example that made use of the proposed fault tolerance features.

Although we are still implementing and evaluating the fault tolerant CAN-CORBA, we strongly believe that additional resource requirements of the fault tolerant CAN-CORBA fall in a reasonable boundary where most CAN based embedded systems could handle. The new CAN-CORBA design demonstrated that it was feasible to use a fault tolerant CORBA in developing distributed embedded systems on real-time networks with severe resource limitations.

7 Acknowledgements

Thanks to the referees and others who provided us with feedback about the paper. Special thanks to Seoul National University RTOS Lab. members who have participated in a series of heated discussions through a year long seminar on this topic.

The work reported in this paper was supported in part by MOST under the National Research Laboratory grant and by Automatic Control Research Center. T.-H. Kim was supported in part by '99 RIET (Research Institute for Engineering and Technology) grant and research fund for new faculty members from Hanyang University, Korea.

References

1. Bosch: CAN Specification, Version 2.0. (1991)
2. Cristian, F.: Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. Distributed Computing, Vol. 4. 1991) 175–187
3. Fetzer, C.: Fail-Aware Publish/Subscribe Communication in Erlang. Proceedings of the 4th International Erlang User Conference. (1998)
4. Object Management Group: Fault-Torerant CORBA Using Entity Redundancy Request For Proposal, OMG Document orbos/98-04-01 edition. (1999)
5. Object Management Group: Minimum CORBA - Joint Revised Submission , OMG Document orbos/98-08-04 edition. (1998)
6. Object Management Group: Fault-Torerant CORBA - Joint Revised Submission, OMG TC Document orbos/99-12-08 edition. (1999)
7. ISO-IS 11898: Road vehicles - Interchange of digital information - Controller Area Network (CAN) for high speed communication. (1993)
8. Kaiser, J., Mock, M.: Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN). IEEE International Symposium on Object-oriented Real-time distributed Computing. (1999)
9. Kim, K., Jeon, G., Hong, S., Kim, S., Kim, T.: Resource Conscious Customization of CORBA for CAN-based Distributed Embedded Systems. IEEE International Symposium on Object-Oriented Real-Time Computing. (2000)

10. Kim, K., Jeon, G., Hong, S., Kim, T., Kim, S.: Integrating Subscription-based and Connection-oriented Communications into the Embedded CORBA for the CAN Bus. IEEE Real-time Technology and Application Symposium. (2000)
11. Oki, B., Pfluegl, M., Siegel, A., Skeen, D.: The Information Bus – An Architecture for Extensible Distributed Systems. ACM Symposium on Operating System Principles. (1993)
12. Rajkumar, R., Gagliardi, M.: High Availability in the Real-Time Publisher/Subscriber Inter-Process Communication Model. IEEE Real-Time Systems Symposium. (1996)
13. Rajkumar, R., Gagliardi, M., Sha, L.: The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. IEEE Real-time Technology and Application Symposium. (1995)