Toward Fault-Tolerant Adaptive Real-Time Distributed Systems

Sam K. Oh and Glenn H. MacEwen

January 1992
External Technical Report
ISSN-0836-0227-
92-325

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6

Document prepared July 24, 1992

Abstract

A monitoring approach to the problem of constructing fault-tolerant and adaptive real-time systems, based on the *fail-signal processor*, is described. Low error detection latency time is a primary goal. A fail-signal processor comprises an *application processor* along with a simple *monitoring processor* that detects abnormal functional or timing behaviour in the application processor; on such a failure the monitor issues a *failure signal* to other fail-signal processors and resets the application processor. The *service-flow graph*, used to specify real-time *services*, shows how a service is decomposed, redundantly designed, and structured to meet time-bounds. Information obtained from service-flow graphs along with run-time information provided by the fail-signal processors permits: (1) forward error recovery from failures in application processors; (2) avoidance or prediction of service timing failures; and (3) reconfiguration with graceful degradation. Avoidance of timing failures is based on adaptive scheduling.

Keywords: Distributed systems. Real-time. Fault-tolerance. Critical systems.

Contents

1 Introduction

Real-time computer systems are required by their environments to produce correct results not only in their values but also in the times at which the results are produced (timeliness). In such systems, the execution times of the *real-time programs* are usually constrained by predefined time-bounds that to-gether satisfy the timeliness requirements. To obtain parallel processing performance and to increase reliability, distributed architectures having processors interconnected by a communications network are increasingly being used for real-time systems. In such *real-time distributed systems*, programs assigned to different processors interact by sending and receiving messages via communication channels.

A failure in a real-time system used for certain applications can result in severe consequences. Such *critical systems* should be designed *a priori* to satisfy their timeliness and reliability requirements. To guarantee the timeliness, one can estimate the worst-case execution times for the real-time programs and check whether they are schedulable, assuming correct execution times. Despite such design, timing failures [1] can still occur at run-time due to unanticipated system or environment behaviours.

For example, a real-time system can fail because of transient overloads caused by excessive stimuli from the environment, or caused by indefinitely blocked programs engaged in resource contention. Furthermore, processors and communication channels can fail, possibly resulting in total system failure. Consequently, design methods for real-time systems that are tolerant of component failures and adaptive to environment behaviours while preserving *predictability* [SR90] are required. A final motivation for such methods is efficiency; in the absence of such methods, significant underutilization of processing resources can occur if estimated program execution times have a safety factor but actual times are much less.

Redundancy, long used in fault-tolerant and adaptive systems, is the provision of resources, such as hardware, software, coding information or time [Joh89], that are not needed if a system can be guaranteed to be fault-free. However, redundancy does not inherently make a system fault-tolerant and adaptive; it is necessary to employ fault-tolerant methods by which the system can tolerate hardware component failures, avoid or predict timing failures, and be reconfigured with little or graceful degradation in terms of reliability and functionality.

Early error detection is clearly important for real-time systems; *error* is an abbreviation for *erroneous system state*, the observable result of a failure. The *error detection latency* of a system is the interval of time from the instant at which the system enters an erroneous state to the instant at which that state is detected. Keeping the error detection latency small provides a better chance to recover from component failures and timing errors, and to exhibit graceful reconfiguration. However, a

[1] When a real-time system or program produces no result, or a result outside of the imposed time-bounds, it is said to have a *timing failure*.

small latency alone is not sufficient; fault-tolerant methods need to be provided with sufficient information about the computation underway in order to take appropriate action when an error is detected. Such information can be obtained during system design and implementation. In current practice, the design and implementation for real-time systems often does not sufficiently address fault tolerance and adaptiveness issues.

In this paper, we introduce a model of real-time distributed systems in which *services* (a service is a non-reflexive partially ordered set of operations) are performed in response to stimuli from the environment, monitored changes in the environment and/or the passage of time. We introduce *service-flow graphs* as design specifications of services. A service-flow graph shows explicitly how a service is decomposed, redundantly designed, and structured to meet a time-bound imposed on that service. We also describe how component services should be allocated to application processors and implemented so that the design specifications as given by the service flow graphs, are satisfied. As a processing component supporting fault-tolerance and adaptiveness in real-time systems, we introduce a compound processor that consists of an *application processor* on which application programs are executed and a simple coprocessor called a *monitoring processor*. We call such a compound processor a *fail-signal processor* because the application processor itself or the monitoring processor issues a failure signal to other fail-signal processors when it detects an abnormal functional or timing behaviour in the application processor. With the information recorded in the service-flow graphs, and certain information provided by the run-time mechanisms (fail-signal processors), real-time distributed systems can use *forward error recovery* to tolerate failures of the application processors, to avoid or predict timing failures of services, and to be reconfigured with graceful degradation. Avoidance of timing failures is based on adaptive scheduling.

2 System Model

We take a real-time distributed system to comprise a set of fail-signal processors interconnected by a communications network. The application processor in each fail-signal processor controls sensors and actuators, and contains a set of software *tasks* and a *run-time kernel*. The tasks, implementations of services specified in a service-flow graph, are scheduled by a local run-time scheduler in the kernel, and interact with their environments (e.g. devices, the network and other tasks) via the kernel. We assume that the network has redundant communication channels and is connected (i.e. no partitioning). We also assume that network messages are received correctly; that is, corrupted messages are detected and discarded.

To overcome transient overloads, *task migration schemes*, i.e. the transfer of tasks in overloaded processors to non-overloaded ones, have been introduced [BSR88, RSZ89, SS91]. In real-time systems, however, the placement of certain tasks is often constrained because they must access sensors and actuators permanently wired to specific application processors. Even when migration is possible it can cause significant processing and communication overheads. Furthermore, migration can make schedulability analysis, crucial for predictability in real-time systems, to be difficult. For these reasons, we address only systems without task migration.

3 Fault-Tolerant and Adaptive Services

Real-time systems should be tolerant of component failures and adaptive to environment changes such as transient overloads. To build such a fault-tolerant and adaptive, yet predictable, real-time system, services should be specified with different levels of required tolerance and acceptable functionality. For each of these different levels a careful analysis of the service's timing behaviours is necessary.

The information obtained from design specifications, as represented in the service-flow graphs, is needed not only during the design of fault-tolerant methods and for predictable behaviour but also for detecting abnormal functional and timing behaviour at run-time.

3.1 Service Characteristics

A service is characterized by an associated set of *input events*, an associated set of *output events* and a timing *upper-bound*. An output event represents the delivery of a result to the environment or to another service; that is, any required communication time occurs before the output event. A service is activated when required input events occur and all other necessary conditions are satisfied. The upper-bound of a service is the maximum allowed interval from the occurrence of the first input event to the occurrence of the last output event. The interval from the occurrence of the first input event to the occurrence of the activation event is called the *waiting time bound*. The interval from the occurrence of the activation event to the occurrence of the last output event is called the *service time bound*. Consequently, the upper-bound is the sum of the waiting time and service time bounds.

Figure 1 shows diagrammatically the characteristics of a service; events associated with a service have unique names although not explicitly shown. The *input condition* of a service, specifying the set of input events necessary for activation, is shown within a triangular *join point* construct at which input events are shown as arriving arrows. The *output condition* of a service, specifying the output events to occur, is shown by a *fork point* construct at which output events are shown as departing arrows. The notation k/n in a join point means that k of n *possible* input events must occur; k^*/n means that *at least* k among n must occur, and k/n in a fork point means that k among n possible output events must occur. A join or fork point associated with only one event can be omitted. Where the number of possible events at a join point or fork point is obvious, k/n and k^*/n can be written as k and k^*.

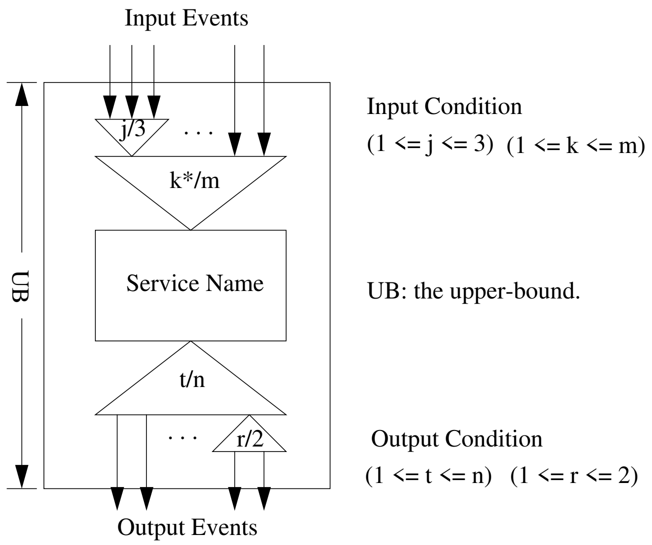


Figure 1: Characteristics of A Service

3.2 Service-Flow Graphs

Depending on the strictness of its upper-bound, a real-time service is categorized as either *hard* or *soft*. In a hard real-time service, outputs produced beyond the upper-bound are not useful, while such outputs in a soft real-time service may be still useful to some, perhaps reduced, degree. Services whose failures, functional or timing, can cause severe consequences, are called *critical* real-time services.

A service can be decomposed into subservices and redundancy incorporated into the design. Such a decomposed service, called a *compound service*, consists of a set of component services, each of which can in turn be decomposed. Each service is specified by a *service-flow graph* (SFG). Of course, intermediate service-flow graphs can be generated during the design of a compound service.

The set of subservices specified in a SFG is partially ordered by the strict *precedes* relation, where 'a *precedes* b' means that b can activate only when a has completed. A service preceding a is called a *predecessor* of a, and a service succeeding a is called a *successor* of a. If there are no intervening services between a and b, a is called the *immediate predecessor* of b, and b is called the *immediate successor* of a.

If a service has an immediate successor, then it has one or more output events each of which causes exactly one input event in that successor. A service with no predecessors is called an *entry service*, and a service with no successors is called an *exit service*. A trivial case of a SFG contains one service, which is both the entry and exit service.

Since there is a finite number of entry and exit services in a SFG, there is a finite number of paths from an entry service to an exit service. For each path in the SFG, one can calculate its *path-bound* as the sum of the upper-bounds of the services on the path. The path with the largest path-bound is the *longest path*. The path-bound of the longest path must not exceed the upper-bound of the compound service.

In calculating the path-bound of a path, the upper-bound of a service may include a non-zero waiting time bound, and the amount of time represented by that waiting time bound may be overlapped with the upper-bound of the immediate predecessor in the path (e.g., services interconnected in parallel). In that case, the waiting time bound must be offset from the path-bound because it is counted twice.

A *primitive service* is one that is not decomposed. Such a primitive service implemented as a program component and assigned to a specific application processor is called a *task*. Each task is scheduled by a local run-time scheduler in its host application processor.

As shown in Figure 2, SFGs allow three types of service interconnection:

1. *serial*,
2. *parallel*,
3. *choice*.

In service G1, showing a serial connection, service S1 can be executed when input event i_1 occurs as the result of output event o_0 from S0. In G2, showing a parallel connection, all n services (S1,S2,...,Sn) can be executed in parallel when activated by their corresponding input events which occur as a result of n output events from Sx. Sy can be executed when n input events occur as a result of n output events from S1 through Sn. The choice interconnection is shown in G3 and G4. G3 shows a set of services (S1,S2,...Sn) among which one can be chosen. On the other hand, G4 shows a service S1 that can be skipped; in other words, *null choices* are allowed in choice interconnections.

In fault-tolerant design, the degree of service redundancy chosen depends on a service's criticality and failure probability. A set of service replicas constitutes a *service group*; each replica becomes a *member* of the group. For example, Figure 3 shows a service-flow graph constructed by duplicating S0 and S1 of service G1 from Figure 2. In this graph, $S0^0$ and $S0^1$ form service group S0, and $S1^0$ and $S1^1$ form service group S1. Since S0 has two members requiring the same inputs, the input environment of G1 may need to be modified to accommodate any necessary synchronization. Similarly, since S1 has two members, each of which issues the same outputs, the output environment of G1 also may need to be modified to accommodate these outputs. Of course, any time delays resulting from these changes should be accounted for in such a re-design.

The service-flow graph allows one to specify up to three different requirements for redundant implementation in a group. First, the group can be required to provide at least k service outputs to activate a succeeding service. Second, the group can be

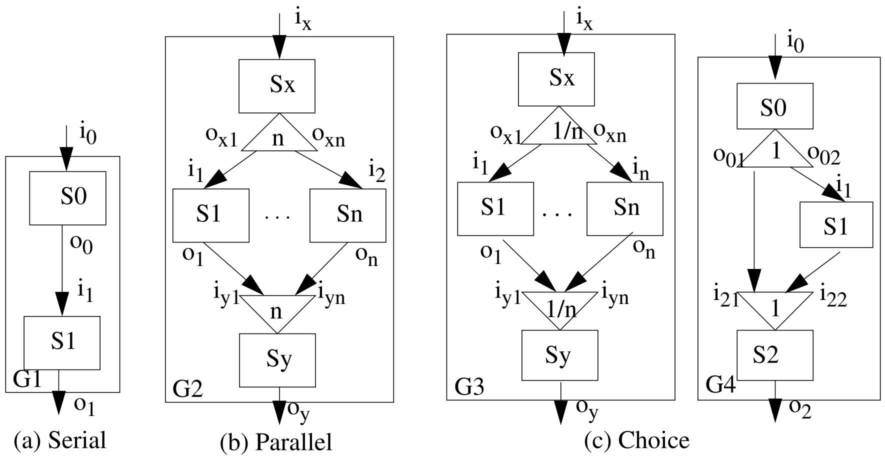


Figure 2: Service Interconnection Types

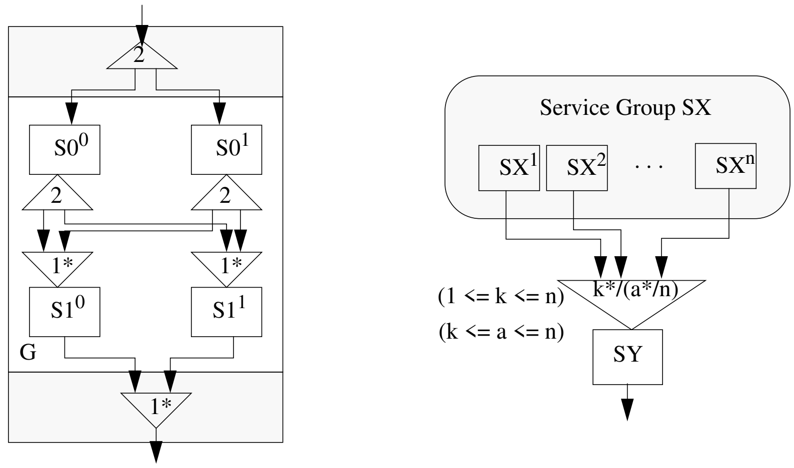


Figure 3: A Service-Flow Graph and A Service Group

required to contain at least a active services with which to provide the outputs. And third, the group can be required to provide a total of n services, of which $n-a$ can be on inactive status (Service status is discussed below.). For example, the diagram shown on the right side of Figure 3 shows a service group SX with n members, and a service SY. The notation $(k^*/(a^*/n))$ in the join-point of SY specifies that at least k input events, among the n possible events, must occur for activating SY. Furthermore, the number of active members must be at least a (clearly, $a \geq k$), and the total number of active and inactive members must be n.

Note, particularly, that not all n group members need to be active. If all members are specified to be active then '(a^*/n)' is written as n. We call k the *output redundancy*, a the *minimum redundancy* , n the *maximum redundancy*, $n-k$ the *potential tolerance degree*, and $a-k$ the *tolerance degree* of group SX. As a further example, the notation 1^* in the join-point of $S1^0$ means that at least one active member of S0 is required. Since service group S0 has two active members ($n = a = 2$), $S0^0$ and $S0^1$, it has a tolerance degree of one.

The number of active members in a service group can fall below the minimum redundancy due to a failure in an application processor to which an active member is assigned. In the case that the group performs a critical service and the output redundancy cannot be relaxed, the specification stands and means that the group must activate inactive members.

Also, the specification can permit the minimum redundancy and/or the output redundancy to be relaxed in the case of a failure. As shown in Figure 4, the non-relaxed and the relaxed redundancies are specified as $k^*/(a^*/n)$ and $rf : j^*/(b^*/n)$ respectively in the join-point of SY, where rf is a threshold value called the *relaxation factor*. When the actual tolerance degree at run-time (the number of active members minus the output redundancy k) becomes less than or equal to rf, the specification of the relaxed redundancy $j^*/(b^*/n)$ applies. The relaxation factor rf must satisfy: $0 \leq rf \leq a-k$.

In order to deal with timing errors, services can be designed to have *multiple versions* of primitive services, each having a different execution time [LC86, KL91] but with a common upper-bound (design diversity), or to have primitive services that can be skipped (imprecise computations) [SLC89, CLL90, LLSY91]. The results produced by such compound services are still acceptable although their functional precision may be degraded.

Such compound services can be specified with choice interconnections. If a compound service has multiple versions, the *longest version* has the greatest execution-plus-communication time, and the *shortest version* has the shortest execution-plus-communication time. The *adaptiveness degree to timing errors* (or simply the adaptiveness degree) of the compound service is the execution-plus-communication time of the longest version minus the execution-plus-communication time of the shortest version. On the other hand, the adaptiveness degree of a compound service with a skippable primitive service is just the upper-bound of the primitive service. In order for run-time schedulers, or tasks themselves, to select suitable versions or to

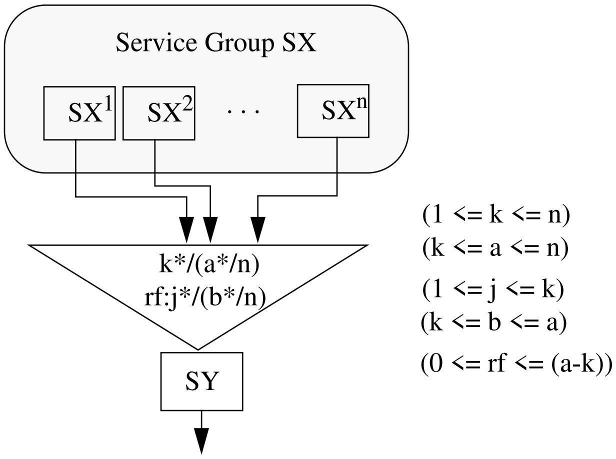


Figure 4: Relaxation of The Redundancy Specification

skip primitive services, the amounts by which actual times deviate from specified bounds can be measured at run-time and provided to the schedulers and tasks.

3.3 Implementation

Once the service-flow graphs of all compound services are produced, the services must be allocated to application processors in a way that satisfies the service-flow graphs' specifications. Several factors constrain the allocation of services. For example, services may require access to resources in specific processors, may have precedence relationships, and may be constrained by communication delays. Services must be allocated so that these constraints are satisfied. Furthermore, if services are not appropriately allocated, even with enough redundancy, the required processor failure tolerance may not be satisfied. Finally, sharing of processors and memories should ideally be minimized where practical to reduce failure correlations and timing uncertainties.

The execution and communication times of tasks are affected by many factors such as program behaviour, input data, the run-time kernel, and processor and communication loading. To ensure predictable behaviour, each task is associated with a *deadline*, the maximum allowed interval from its activation to its completion. To assign appropriate deadlines it is necessary to estimate task execution and communication times. We assume that *best-case* and *worst-case* execution and communication times can be obtained with known techniques [PK89, PS91, SHH91].

In summary, each task is associated with the following timing parameters: a deadline, best-case execution and communication times, and worst-case execution and communication times. It is important to note here that the execution time includes the waiting time for task activation.

After the resource allocation of primitive services, a *task allocation and communication map* (TACM) is produced for each service-flow graph. The TACM shows, for each primitive service (hence task), the processor to which it is assigned, timing parameters, input and output conditions, and communication delays.

The TACM of a service-flow graph also shows how component tasks of the graph are interconnected as well as the *attributes* of each task. An *entry* (*exit*) task implements an entry (exit) primitive service; other tasks are *intermediate*. A *non-interacting* task implements a trivial service; other tasks are *interacting*. A non-interacting task is both an entry and exit task. A *multi-version* task implements a primitive service in a set of multi-version services. A *skippable* task implements a skippable primitive service. Finally, a task producing low quality, yet acceptable results is an *imprecise* task [LLSY91].

A set of service replicas constitutes a *service group*. Each service group is described by its *service group membership* (SGM) information; each member is associated with its host processor, and is assigned a *member status* of which there are four types:

- Active (A),
- Standby (S),
- Passive (P),
- Failed (F).

Both active and standby members are schedulable so that they can maintain application state information. However, outputs produced by standby members are not used while those of active members are. Passive members are not schedulable. When a member of a service group fails (or its host application processor fails), its status becomes Failed (F). Since all tasks are statically allocated and cannot be migrated, service membership information is fixed except for the status of members. Members of a service group are not necessarily tasks; a member can be a partially-ordered set of tasks. For simplicity, we assume that members are single tasks.

Each application processor is loaded with a set of tasks including the tasks' attributes, input/output conditions, timing parameters and the SGM. The monitoring processor is loaded with information describing each service to which monitored tasks belong (TACMs and SGMs). The table in Figure 5 shows service group membership with design and implementation information.

Although a service group may perform a critical service, not all members need to be scheduled with their timing constraints. For example, standby members need not meet deadlines because outputs are not needed, and passive members are not scheduled. A standby member of a group can become active due to the failure of an active member; since standbys maintain state information such reconfiguration is made easier. If a passive member must become either standby or active, state information must be provided.

Group	Member	Processor	Status
GS0	A0	P0	A
	A1	P1	A
GS1	B0	P0	A
	B1	P1	S
	B2	P2	P
GS3	C0	P3	A
	C1	P4	F
.	.	.	.
.	.	.	.

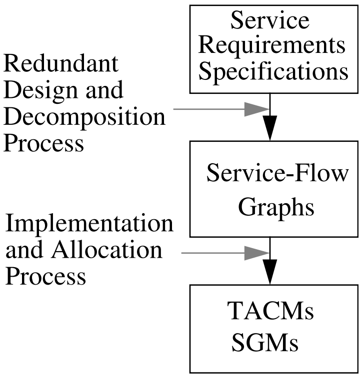


Figure 5: A Table of SGM and Design and Implementation Information

Since relaxation can be specified in a service-flow graph, different task sets can exist in each application processor. Consequently, it is necessary to analyze schedulability for each possible task set.

4 Run-Time Support

Continuous monitoring seems necessary to enable detection of abnormal functional or timing behaviour with small error detection latencies. Unfortunately, most such approaches currently proposed monitor either functional or timing behaviour but not both, and are intended for uniprocessors or multi-processors. An integrated approach that monitors both kinds of behaviour for real-time distributed systems is needed.

4.1 Related Work

4.1.1 Functional Monitoring

A transient or permanent hardware fault in an application processor can cause a *control-flow error*, an invalid sequence of instructions. Since control-flow errors can cause arbitrary behaviour (Byzantine failures: [LSP82, SD83]), *watchdog-based monitoring* techniques [Lu82, MM88, WS88] [2] have been introduced to detect them. Using these techniques, the control-flow behaviour of an application processor is monitored by an attached simple co-processor called a watchdog.

Watchdog-based techniques generally involve two phases: a *setup phase* and a *checking phase*. During the setup phase, usually at compile-time, the watchdog is provided with certain information about the programs to be executed. During

[2] Experiments based on *fault injection* [SS87, MM88, GKT89], i.e. artificial introduction of faults into the components of application processors, show that roughly 80% of the injected faults cause control-flow errors.

the checking phase at run-time the watchdog continuously monitors the application processor, collects corresponding run-time information, and detects errors via any discrepancies between the information collected and the information provided.

Without incurring much extra cost, watchdog-based techniques allow low detection latency for control-flow errors. However, they don't seem to be effective at detecting invalid data values [SS87, MM88, GKT89], which may also result from transient or permanent hardware faults. Real-time systems usually contain critical variables for which invalid values may cause severe consequences. Such variables must be checked for acceptable in-range values. Furthermore, an application processor can respond too late or not at all due to resource contention, an infinite program loop or a processor crash. Techniques dealing with such failures are needed.

Current watchdog-based methods are intended for programs running on a uniprocessor or multiprocessor. In real-time distributed systems, in which tasks require intra-group coordination, tasks often maintain replicated state information that must be consistent. Since a processor failure can threaten the consistency of such information, fast notification of a failed processor to other non-failed processors is necessary.

4.1.2 Timing Monitoring

Haban and Shin [HS90] propose a monitoring approach in which each task is divided into a set of disjoint program segments according to the syntactic structure of the task. Assuming knowledge of the maximum number of loops within each of these segments, the worst-case pure execution time for each segment is estimated. In their approach, a specially designed processor, called TMP, measures the true execution time and resource sharing delay of tasks at run-time. The measured results enable the run-time scheduler, a part of TMP, to schedule tasks adaptively.

Gopinath and Gupta [GG90] propose a 'compiler-assisted' approach in which the compiler examines the code of each application task and partitions it into *typed* segments. The type of each segment is determined by the combination of two criteria: predictability and monotonicity. A segment is predictable if it has a fixed execution time; otherwise, where the execution time is determined by input data, it is unpredictable. A segment is monotonic if its output quality is monotonically improved as it is executed longer; otherwise it is non-monotonic. During compilation, program segments are re-ordered in such a way that unpredictable and/or non-monotonic segments can be executed first. At the end of each unpredictable segment, measurement code is inserted to measure the actual execution time of the segment; a time deviation is calculated by subtracting this measured time from the estimated worst-case execution time of the segment. We call this estimated execution time the *segment time bound*. If the accumulated segment time deviations of a task indicate a timing error, the run-time scheduler takes recovery action by changing the loop bounds of monotonic segments. Tasks that cannot be recovered are aborted.

In a real-time distributed system, a compound service does not necessarily have a timing failure when a component task fails since timing errors caused by preceding tasks can be compensated in succeeding tasks. A compound service can comprise a set of service replicas to tolerate application processor failures; failures of component tasks, functional or timing, do not necessarily result in failure of the service. Unfortunately, current timing behaviour monitoring methods do not deal with functional failures. Furthermore, they only deal with non-interacting tasks, i.e. implementations of trivial services.

4.2 Fail-Signal Processor Monitoring

A fail-signal processor consists of an application processor (AP) and a monitoring processor (MP). The MP, which is similar to the monitor used in current watchdog-based methods, monitors control-flow behaviour in the AP. The AP is assumed to have self-checking capabilities; value errors such as invalid input data can be detected. When the MP detects a control-flow error, it resets the AP and issues a control-flow failure signal. Similarly, when the AP detects a value error it issues a value failure signal, and resets itself if necessary. These failure signals are sent as network messages to be used by other fail-signal processors in carrying out recovery and reconfiguration. A reset AP enters a self-diagnosis mode and rejoins the system only after repair or self-determined correctness and it receives approval from the other fail-signal processors. The MP also supports the restarting and reconfiguration of the AP.

A task can fail due to an infinite program loop, or a host AP crash. Since these failures, like control-flow errors, cannot be detected by the AP itself and can result in a service failure, we give the responsibility for failure prediction and detection to the MP. When a service failure is predicted, a strategy determined during design time can be enforced. Depending on the service criticality, the entire system may be stopped, some analog or human alternative can assume control, or the service may be aborted to provide more time for other services.

A *time deviation* is any difference between a specified time bound and the corresponding run-time interval. A negative time deviation indicates a timing error. The run-time kernel in each AP provides time deviation information to its scheduler and application tasks. The kernel also supports its MP by notifying the MP of certain events necessary for predicting service failures. The MP makes use of time deviation information in messages received via the network.

Figure 6 shows two fail-signal processors interconnected by a network. The functions performed by each are listed and the failure signal paths are shown with the labels FSIG(V,...) for value failures and FSIG(CF,...) for control flow failures. The label MSG(τ,..) represents a message containing time deviation information. The internal AP/MP interface is also shown.

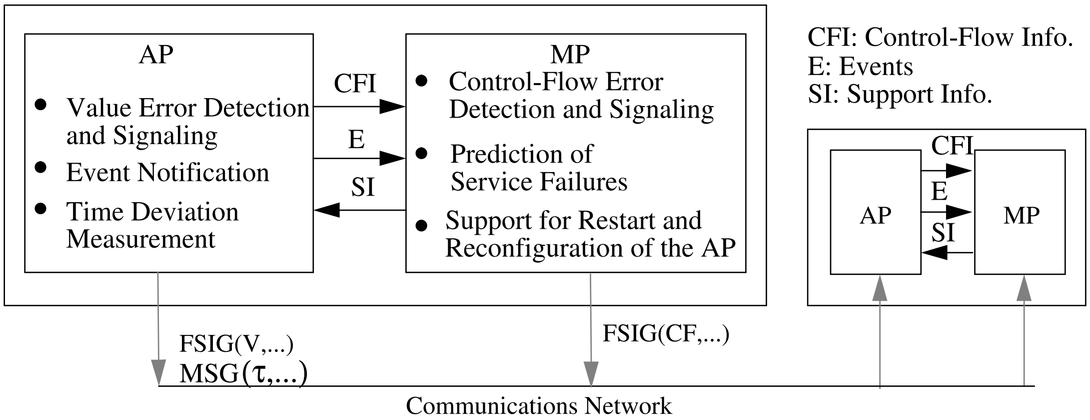


Figure 6: Interconnection of Two Fail-Signal Processors

4.3 Design Goals

In designing the internal and external interfaces of a fail-signal processor, we have three primary goals:

- Non-intrusive monitoring: The MP should not normally interfere with the services performed by the AP; i.e. the MP receives run-time information from the AP through their direct interface or from the network.
- Minimal overhead: Overhead within the AP should be minimized.
- Minimal modification: Required changes to standard AP hardware and software should be minimized.

With this approach, a failure in an MP should not affect services performed by its AP, even though fault tolerance capabilities may be lost or degraded to a certain degree. An MP is assumed to be designed for *fail-stop* semantics.

4.4 Interfaces

The run-time kernel in an AP is the interface between an application task and its environment. This section describes how the kernel for an AP is structured, measures time deviations, notifies the MP of certain events, and reacts to detected errors and failure signals. We also briefly describe criteria that could be used by the MP for prediction. Since there are many watchdog-based methods for detecting control-flow errors [Lu82, MM88, SS87, SM90, MS91], we assume that a suitable one is used. However, it is an interesting question as to which of such methods are best suited for the fail-signal processor architecture.

4.4.1 Run-Time Kernel

The scheduler in an AP attempts to select tasks at run-time so that each task deadline can be met. One complication in predicting schedulability is that there can be significant delays between event occurrences and their recognition. In conventional real-time systems, event handling programs such as interrupt handlers are usually executed immediately even though such quick executions may not be necessary. Consequently, delays may occur in recognizing other events. The run-time kernel should be designed to recognize event occurrences as quickly as possible.

As an approach to fast recognition of event occurrences we structure the AP software into layers. See Figure 7. The applications layer comprises application tasks, and perhaps some system tasks that can be treated similarly. The kernel has three layers:

- Event recognition layer.
- Dependability management layer.
- Adaptive scheduling layer.

The event recognition layer is responsible for *event recording* and *time management*. The event recording mechanism records the times of input and output event occurrences such as device interrupts, timeouts, shared variables updates and messages sends and receives. The time of occurrence of a message send (receive) is defined to be the time at which the last data byte is transmitted (received). When a message is sent its occurrence time is appended to the message.

The dependability management layer includes *time deviation measurement*, *event notification*, and *recovery and reconfiguration* functions; details are described later. The adaptive scheduling layer includes an adaptive scheduling algorithm that attempts to schedule tasks by making use of information obtained from the service-flow graphs and the time deviation information. The hardware is assumed to provide a high-resolution clock.

The efficiency and predictability of a real-time system can be improved by physically separating the kernel from the applications. For example, a shared-memory multiprocessor architecture can be used for this purposes. One processor could be specially designed for the kernel, with the others used for application tasks. With this architecture, unnecessary interruptions of application tasks can be minimized, and the number of context-switches can be significantly reduced. Also, the timing behaviour of the application tasks and the kernel become more predictable. The fail-signal processor can be easily adapted to such an architecture.

4.4.2 Time Deviation Measurement

A *measurement point* is any instant at which a time deviation in a service is measured. For example, earliest input event and activation event occurrences are measurement points for a service. At each measurement point of a service the kernel calculates a time deviation.

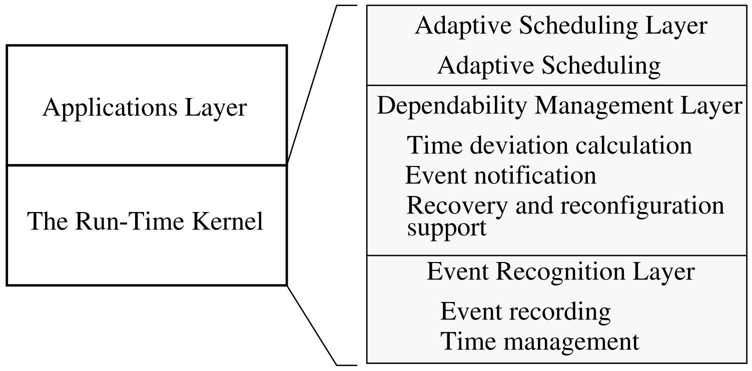


Figure 7: Layered Approach: The Applications Layer and The Kernel Layers

We now describe how to measure the time deviation of a service using as an example the compound service G shown in Figure 8. We assume that an input event i0 occurs at $t0$, and the time deviation of G is initially zero. Since task S0 has no other input event, the activation time is $t0$. We denote the time deviation of G at $t0$ as $\tau(t0)$. The kernel associates with S0 the activation time $t0$, the service time bound SB0, and the time deviation $\tau(t0)$.

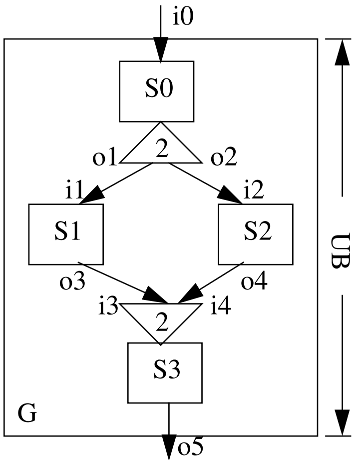
When S0 issues a message to S1 (event o1), the kernel inserts ($t0$+SB0) and $\tau(t0)$ into the message, and appends the message transmission time t_{o1}. Similarly, when a message is issued to S2 (event o2), the kernel inserts ($t0$+SB0), $\tau(t0)$, and t_{o2} into the message.

When the kernel in AP1 recognizes an event i1 (i.e., a message from S0), it records, with reference to its local clock time, the occurrence time $t1$. $t1$ becomes the activation time of S1. At $t1$, the kernel measures the time deviation for S0, which is: ($t0$+SB0) - (t_{o1}+δ_{i1})
where δ_{i1} is the message transmission delay.

If the network does not guarantee fixed transmission times, a clock synchronization assumption seems necessary. Assuming clock synchronization, we obtain δ_{i1} by ($t1$-t_{o1}). If fixed message transmission times are guaranteed, we can calculate the actual transmission delay δ_{i1} by multiplying the unit message transmission time by the length of the message. Problems due to clock drift and the period of clock synchronization are ignored here.

Once the time deviation of S0 is obtained, the kernel obtains $\tau(t1)$, the time deviation of service G at $t1$, by adding the time deviation of S0 to $\tau(t0)$. When task S1 issues an output message (event o3) to S3, the kernel inserts ($t1$+SB1), $\tau(t1)$, and t_{o3} into the message. The kernel in AP2 performs similarly.

Task S3 is different; it can be activated only when events i3 and i4 both occur. Assume that i3 occurs first at time $t3$ (i.e., the earliest input event) and i4 occurs next at time $t4$. At $t3$, the kernel calculates the time deviation for S1, and obtains $\tau(t3)$ by adding it to $\tau(t1)$. At $t4$ (i.e. at the activation time of S3), the kernel



WTB: Waiting Time Bound
STB: Service Time Bound
D: Deadline

Task-id	Proc-id	WTB	STB	D
G.S0	AP0	0	SB0	d0
G.S1	AP1	0	SB1	d1
G.S2	AP2	0	SB2	d2
G.S3	AP3	WB3	SB3	d3

Figure 8: A Compound Service With Four Primitive Services

calculates the waiting time deviation by (WB3 - ($t4$-$t3$)), i.e. waiting time bound minus the actual waiting time of S3, and obtains $\tau(t4)$ by adding it to $\tau(t3)$. The current value of $\tau(t4)$ shows the amount of the time deviation of service G up to the activation of S3.

It was noted previously that a service consisting of a single non-interacting task can be partitioned into segments. Similarly, tasks occurring near the end of a compound service can be partitioned into segments if more frequent measurements are desirable. A kernel call is inserted at the end of each segment, allowing the kernel to detect segment completion time (a *segment event*) and to update time deviation information. The kernel also notifies the MP of the segment event. Of course the segment time bound must have been provided to the kernel and the MP prior to run-time.

4.4.3 Event Notification

The run-time kernel supports the MP by notifying it of certain events necessary for service failure prediction. For such notifications, a unidirectional high-speed bus from the AP to the MP is used. Events transmitted to the MP are:

- earliest input event (Sid,Tid)
- activation event (Sid,Tid).
- rejection event (Sid,Tid).
- segment event (Sid,Tid,Bid).
- completion event (Sid,Tid).

where Sid, Tid and Bid respectively represent the identification of an associated service, primitive service (task) and segment.

Under certain conditions, the AP may not be able to activate a task or continue a task execution: in the case of, for example, a failed task or a passivated task. The kernel informs the MP of such a happening by sending a rejection event. The kernel also notifies the MP of task completion by sending a completion event. If the MP does not receive notification of the activation event, segment event, or completion event within a preset time limit after the earliest input event, it assumes task failure and hence failure of the service to which the task belongs.

4.4.4 Failure Prediction

An *evaluation point* of a service is an instant of time at which its status is evaluated. Any occurrence of an earliest input event, an activation event, a segment event, or a completion event for a task can be an evaluation point of the service to which the task belongs. The MP evaluates service status when notified of these events. However, it is not necessary to evaluate services at all such points. Rather, the determination of evaluation points can be made at design time.

At an evaluation point i, the status of a compound service is characterized by the following four parameters:

- SA: Safety Allowance.
- $\tau(i)$: Time Deviation.
- RUB(i): Remaining Upper-Bound.
- RT(i): Remaining Waiting and Service (execution-plus-communication) Time.

where the *safety allowance* of a compound service is defined to be its upper-bound minus the path-bound of the longest path. For convenience, we define the laxity of a service at an evaluation point i as:

$$\mathrm{L(i) = RUB(i) - RT(i)}$$

By using the estimated best-case and the worst-case service times of each task obtained during the implementation and analysis, we can calculate two different laxities:

- BL(i) = RUB(i) - BRT(i).
- WL(i) = RUB(i) - WRT(i).

where BRT(i) is the remaining best-case waiting and service time and WRT(i) is the remaining worst-case waiting and service time. We call BL(i) the *best-case laxity*, and WL(i) the *worst-case laxity*.

A non-negative value of $\tau(i)$ indicates that the service is processing faster than expected. In this case, the amount of time represented by $\tau(i)$ can be used for other services. On the other hand, a negative value of $\tau(i)$ indicates that the service has incurred a timing error.

In evaluating the *severity* of a timing error, the MP uses the following criteria:

1. $abs(\tau(i)) \geq BL(i)$: Failure Zone.
2. $WL(i) \leq abs(\tau(i)) < BL(i)$: Uncertainty Zone.
3. $SA < abs(\tau(i)) < WL(i)$: Slight Deviation Zone.
4. $abs(\tau(i)) \leq SA$: Safety Zone.

where $abs(\tau(i))$ represents the absolute value of $\tau(i)$. If a service is in the failure zone, it will fail even though it is continued. If a service is in the uncertainty zone, it is difficult to predict its future timing behaviour (although statistics based on previous run-times may be useful). If a service is in the slight deviation zone it is unlikely to fail. If a service is in the safety zone it is, within the assumptions of the design, certain not to fail.

At the current state of the development of our ideas, we propose to evaluate a task, and hence the service to which it belongs, only at its earliest input event. At an evaluation point i, assuming that the task is not an exit task, the sum of $\tau(i)$, UB(i) and BL(i+1) is used as the time limit for judging failure of the service to which the task belongs; UB(i) is the upper-bound of the primitive service corresponding to the task. If the task is not completed within this limit, the service is assumed to have failed. We are currently investigating other prediction criteria for evaluating service status.

4.4.5 Recovery and Reconfiguration

The kernel's dependability management layer deals with value errors, which can be detected by the kernel itself (e.g., message validity checks) or in application tasks. When a task having a value error is detected, the kernel aborts it, changes its status to 'F' and sends a rejection event to the MP. The kernel then identifies the task's group members and immediate successors and multicasts a failure signal FSIG(V,Proc-id,Failed Task-id,...) to the AP's containing them. If relaxed redundancies are specified for the group and relaxation is required, this is indicated in the failure signal.

The kernel also reacts to failure signals. Assume that a task, an active member of a service group, has failed. Once the kernel is notified of the failure, it changes the status of the failed member to 'F' if its host AP contains a member of the group. If the failure signal indicates relaxation, the kernel appropriately changes the redundancy specification.

In order to maintain the minimum redundancy specified for the group, the kernel attempts to change the status of a non-failed passive or standby member to 'A'. For this reconfiguration, the kernel may be required to force the failure of certain tasks. In turn, the redundancy specifications of the service groups to which these failed tasks belong may need to be relaxed. Since this process cannot be carried out alone, the kernel interacts with other fail-signal processors, including the one containing the failed task.

When an MP detects a control-flow error, it resets its AP and multicasts a failure signal FSIG(CF,Proc-id,...) to all AP's containing at least one related task. A related task is any member of a group containing a task in the failed AP, or any successor of a task in the failed AP. Broadcasting can be performed *group-by-group*; in this case the multicasting order may depend on the relative importance of the groups.

When the number of notified task failures at an MP reaches a predetermined threshold value, it resets the AP and multicasts a failure signal. This signal is treated in the same way as a failure signal due to a control flow error in the AP. The MP also supports the failed AP when it restarts and rejoins the system.

5 Conclusion

Fault-tolerant and adaptive real-time systems not only require low latency support for error detection, but also need application information to permit failure prediction, failure avoidance, and appropriate recovery action. We have introduced a monitoring approach based on fail-signal processors, which allows continuous observation of the functional and timing behaviour of application processors. The service-flow graph is used as the vehicle for specifying services. By incorporating top-down design and fault-tolerant methods such as design diversity to deal with timing errors, and design redundancy to deal with functional errors, a designer can specify explicitly how a service is decomposed, redundantly designed, and structured to meet its time-bound. Information obtained from the service-flow graphs, along with run-time information from the fail-signal processors, permits forward error recovery after failures in application processors, prediction and avoidance of timing failures, and reconfiguration with graceful degradation.

Recently, attention has been paid to the monitoring approach for debugging and performance monitoring of multiprocessor and distributed systems [LSMC90, Cas91]. The fail-signal processor, when used for multiprocessor and/or distributed systems, can also be useful for system testing, debugging, performance measurement, and validation.

The research reported here is a part of a larger project to investigate methods and tools for designing real-time distributed systems [BCMM86, CMM87]. We are currently addressing service failure prediction with more detailed prediction criteria, and adaptive scheduling algorithms that can efficiently make use of design time and run-time information. Dynamic reconfiguration algorithms are also under investigation.

References

[BCMM86] C. Belzile, M. Coulas, G. H. MacEwen, and G. Marquis. Rnet: A hard real-time distributed programming system. *Proceedings of the Real-Time Systems Symposium*, pages 2–13, December 1986.

[BSR88] S.R. Biyabani, J.A. Stankovic, and K. Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. *Proceedings of the Real-Time Systems Symposium*, pages 152–160, December 1988.

[Cas91] T.L. Casavant. Panel session: Debugging and performance monitoring for distributed systems: Problems and prospects. *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 378–383, May 1991.

[CLL90] J.Y. Chung, J.W.S. Liu, and K.J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1174, September 1990.

[CMM87] M. Coulas, G. H. MacEwen, and G. Marquis. RNet: A hard real-time distributed programming system. *IEEE Transactions on Computers*, 36(8):917–932, August 1987.

[GG90] P. Gopinath and R. Gupta. Applying compiler techniques to scheduling in real-time systems. *Proceedings of the Real-Time Systems Symposium*, pages 247–256, December 1990.

[GKT89] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. *The 19th Symposium on Fault-Tolerant Computing*, pages 340–347, June 1989.

[HS90] D. Haban and K.G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. *IEEE Transactions on Software Engineering*, 16(12):1374–1389, December 1990. Also in Real-Time Systems Symposium 1989, pages 172-181.

[Joh89] B. W. Johnson. Design and analysis of fault tolerance digital systems, 1989.

[KL91] K.B. Kenny and K.J. Lin. Building flexible real-time systems using the flex languages. *IEEE Computer*, 24(5):70–78, May 1991. Also shown in technical report UIUCDCS-R-90-1634 University of Illinois at Urbana-Champaign.

[LC86] A.L. Liestman and R.H. Campbell. A fault-tolerant scheduling problem. *IEEE Transactions on Software Engineering*, 12(11):1089–1095, November 1986.

[LLSY91] J.W.S. Liu, K.J. Lin, W.K. Shih, and A.C. Yu. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991. Also as a tech. report UIUCDCS-R-90-1628, Dept. of Computer Science, University of Illinois-Urbana.

[LSMC90] J.E. Lumpp, Jr., H.J. Siegel, D.C. Marinescu, and T.L. Casavant. Specification and identification of events for debugging and performance monitoring of distributed multiprocessor systems. *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 476–483, June 1990.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[Lu82] D.J. Lu. Watchdog processors and structural integrity checking. *IEEE Transactions on Computers*, 31(7):681–685, July 1982.

[MM88] A. Mahmood and E.J. McClusky. Concurrent error detection using watchdog processors- a survey. *IEEE Transactions on Computers*, 37(2):160–174, February 1988.

[MS91] Henrique Madeira and Joao G. Silva. On-line signature learning and checking. *Second International Conference on Dependable Computing for Critical Applications*, pages 170–177, February 1991.

[PK89] P. Puschner and CH. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, pages 159–176, 1989.

[PS91] C.Y. Park and A.C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, May 1991.

[RSZ89] K. Ramamritham, J.A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, 38(8):1110–1123, August 1989.

[SD83] H. R. Strong and D. Dolev. Byzantine agreement. *Digest of Papers from COMPCON, IEEE*, pages 77–81, Spring 1983.

[SHH91] A. Stoyenko, V. Hamacher, and R. Holt. Analyzing hard-real-time programs for guaranteed schedulability. *IEEE Transactions on Software Engineering*, 17(8):737–750, August 1991.

[SLC89] W.K. Shih, J.W.S. Liu, and J. Chung. Fast algorithms for scheduling imprecise computations. *Proceedings of the Real-Time Systems Symposium*, pages 12–19, December 1989.

[SM90] N.R. Saxena and E.J. McClusky. Control-flow checking using watchdog assists and extended-precision checking. *IEEE Transactions on Computers*, pages 554–559, April 1990.

[SR90] J.A. Stankovic and K. Ramamritham. Editorial: What is predictability for real-time systems. *The Journal of Real-Time Systems*, 2:247–254, 1990.

[SS87] M.A. Schuette and J.P. Shen. Processor control flow monitoring using signatured instruction streams. *IEEE Transactions on Computers*, 36(3):264–276, March 1987.

[SS91] N. Shivaratri and M. Singhal. A transfer policy for global scheduling algorithms to schedule tasks with deadlines. *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 248–255, May 1991.

[WS88] K. Wilken and J.P. Shen. Continuous signature monitoring: Efficient concurrent-detection of processor control-flow errors. *Proceedings on International Test Conference*, pages 914–925, September 1988.