A Novel Fault Tolerant Scheduling Technique In Real-Time Heterogeneous Distributed Systems Using Distributed Recovery Block

Bibhudatta Sahoo
Senior Lecturer
Department of CSE, NIT Rourkela

Aser Avinash Ekka
JRF (CS-HDS)
Department of CSE, NIT Rourkela

Abstract
Fault-tolerance is an important requirement for real-time distributed system, which is designed to provide solutions in a stringent time constraint. This paper considers fault-tolerant scheme on heterogeneous multi-component distributed system architecture using a software technique based on Distributed Recovery Block (DRB). The experiment shows that, the proposed scheme based on DRB with Random-EDF heuristic tolerates about 10% to 20% number of permanent failures and an arbitrary number of timing failures.

1. Introduction
Distributed heterogeneous computing is being widely applied to a variety of large size computational problems. These computational environments are consists of multiple homogeneous computing modules, these modules interact with each other to solve the problem. Real-time Distributed systems (RTDS), such as aircrafts and automobiles, nuclear, robotics, and telecommunication, require high dependability, where system failures during execution can causes catastrophic damages. These systems must function with high availability even under hardware and software faults.

No matter how meticulously error avoidance and error detection techniques are used, it is virtually impossible to make a practical system entirely error free. Therefore, to achieve high reliability, even in situation where errors are present, the system should be able to tolerate the faults and compute the correct results this is called fault-tolerance. Fault-tolerance can be achieved by carefully incorporating redundancy.

One major advantage of distributed systems is to tolerate individual component failure without terminating the entire computation [3,4,5]. Research in fault-tolerant distributed computing, aims at making distributed systems more reliable by handling faults in complex computing environments. Moreover, the increasing dependence of different services on real time heterogeneous distributed system has led to an increasing demand for dependable systems, systems with quantifiable reliability properties. The faults in a distributed computing system may appear either in the hardware or in the software and they can be classified as being permanent, intermittent or transient [13,14,15]. In fault-tolerant Real Time Distributed systems, detection of fault and its recovery should be executed in timely manner so that in spite of fault occurrences the intended output of real-time computations always take place on time. For fault tolerant technique detection, latency and recovery time are important performance metrics because they contribute to node downtime. A fault tolerant technique can be useful, in RTDS if its fault detection latency and recovery time are tightly bounded. When this is not feasible, the system must attempt the fault tolerance actions that lead to the least damages to the applications mission and the systems users.

Task scheduling techniques can be used to achieve effective fault tolerance in real time systems [5, 3]. This is an effective technique, as it requires very little redundant hardware resources. Fault tolerance can be achieved by scheduling additional ghost copies in addition to the primary copy of the task. We present our approach based on software redundancy to tolerate permanent and timing failures. We propose to use distributed recovery block to perform software redundancy, where a given input a task (Ti) is augmented with a redundancies. Then, operations and data-dependences of Ti can be distributed and scheduled on a specified target distributed architecture (G) to generate a fault tolerant distributed schedule.

2. Real Time Distributed Computing System
We consider a heterogeneous distributed computing system (HDCS) consists of a set Ω of n Nodes (uniquely addressable computing entity)\{P_1, P_2, ..., P_n\}, \text{P}_i=\text{(A}_i, \text{ε}_i), \text{where A}_i \text{ is the set of tasks in the queue of P}_i \text{, ε}_i \text{ is the fixed execution rate. Each processor was assumed to have different execution rate measured in MFLOPS/s}[6] and they are connect with each other using bi-directional point-to-point communication links . In heterogeneous distributed system a task \text{τ}_j has different computation time which is measured by \text{τ}^n_j which represents the time of task \text{τ}_j on processor \text{P}_i where \text{l i} \leq n \text{ and l j} \leq N. The processors of the distributed system are heterogeneous and the availability of each processor can vary over time (processors are not dedicated can may have other tasks that partially use their resources). We have extended the
model mentioned in [8] where each processor has a primary and backup queue as shown in Figure 1.

![Centralized Queue](image)

Figure 1 Example of FT RTDS architecture for 2 nodes

### 3. Periodic Task Model

A periodic task is characterized by its period of release and they are to be executed exactly once in every period. Hence, the period of the task is also its deadline. We denote the set of tasks in the application by the set $T = \{T_1, T_2, T_3, \ldots, T_d\}$ where a task $T_i$ is periodic. The basic task model is $\Pi$ modeled by a set of $N$ periodic tasks:

$$
\Pi = \{T_i = (P_i, D_i, C_i) \mid 1 \leq i \leq N\} \quad (1)
$$

where $P_i$ is the period of the task. Each task is released every $P_i$ time units. The difference in time between the arrivals of two consecutive instances of a periodic task is always fixed and is referred to as period of that task. Although the inter arrival times of instances of a periodic task are fixed, the inter release time may not be. $D_i$ is the relative deadline, the period of time after the release time within which the task has to finish. Tasks can have arbitrary deadline. $C_i$ is the worst-case execution time of the task.

We consider each Task $T_i$ is assumed to consist of a set of subtasks, which execute “serially”. For convenience, we denote the set of subtasks of task $T_i = \{s_{i1}, s_{i2}, \ldots, s_{ik}\}$ as shown in Figure 2. where $\otimes$ is the intermediate deadline [7].

$$
s_{i1}, s_{i2} \otimes k\%, s_{i3}, s_{in}
$$

Figure 2. Subtasks and intermediate deadline

### 4. Previous Work

Occurrence of a fault during execution in any stem requires, extra time to handle fault detection and recovery. In case of real time system in particular, it is essential that extra time be considered and accounted for prior to execution. The methods used for real time fault tolerant system, must consider the number and types of fault subjected to without violating the timing constraints. Fault tolerance has typically been approached from a hardware standpoint, with multiple replicas of essential applications running on separate hardware components mostly in parallel fashion. In the area of real time distributed systems, a fault-tolerant scheduling strategy is described in [9][6][11][12]. The requirements of a fault tolerant scheduling algorithm in real time distributed systems are described in [2][13]. A successful fault tolerant primary/backup algorithm with the dynamic EDF algorithm for multiprocessors running in parallel and executing real-time applications [13, 1]. One of the major technique for achieving fault tolerance is replication but the level of replication is chosen depending on the desired fault tolerance required [15] discusses a replication control mechanism in distributed real time database system. Software based fault tolerant application using a single version scheme (SVS) is described in [37]. A middleware based MEAD infrastructure aims to provide a reusable, resource-aware real-time support to applications to protect against crash, communication, partitioning and timing faults are discussed in [16]. Kim also outlines the other middleware techniques of fault tolerance. Fault tolerant techniques implemented by means of scheduling are discussed in [17, 18, 19].

### 5. DRB Scheme

The Control Implementation Structures used in this paper is DRB: Distributed Recovery Block. In this paper we have outlined the two requirements for DRB variant i.e. Primary-Backup Fault tolerant algorithm in RTDS is that (1) the execution of backup versions should not hinder the execution of the primary version of the tasks (2) when the primary task fails to meet its deadline the backup instance should then be executed but it should be executed from the point of last correct subtask executed by the primary version. This paper we propose new an extended distributed recovery block based fault tolerant scheduling algorithm for real time tasks. Our algorithm ensures that the parallel execution of backup task works better in case of transient overload and handles both permanent and timing fault.

The distributed recovery block (DRB) scheme is an approach for realizing both hardware fault tolerance and software fault tolerance in real-time distributed and/or parallel computer systems. The underlying design
philosophy behind the DRB scheme is that a real-time distributed or parallel computer system can take the desirable modular form of an interconnection of computing stations, where a computing station refers to a processing node (hardware and software) dedicated to the execution of one or a few application tasks [14]. The idea of the distributed recovery block (DRB) has been adapted from [2][13]. Recovery block consists of one or more routines, called try blocks here, designed to compute the same or similar result, and an acceptance test which is an expression of the criterion for which the result can be accepted both in terms of correctness and timing constraint. For the sake of simplicity a recovery block consists of only two try blocks, i.e. primary and backup [9, 1]. The error processing technique used is acceptance test, that is parallel between node pairs but sequential in each node with complexity $O(n)$.

6. The Fault Tolerant Scheduling scheme

The main idea of software fault tolerance is to contain the damage caused by software faults. Several techniques that can be used to limit the impact of software faults (read bugs) on system performance. Efforts to attain software that can tolerate software design faults (programming errors) have made use of static and dynamic redundancy approaches similar to those used for hardware faults[3,22]. Techniques involved in achieving software fault tolerance are: (i) timeouts, (ii) audits, (iii) exception handling, (iv) task rollback, (v) incremental reboot, (vi) voting, (vii) n-version programming, (viii) recovery-block approach, and (ix) algorithm based fault tolerance [21]

In fault-tolerant real time distributed systems, detection of fault and its recovery should be executed in timely manner so that in spite of fault occurrences the intended output of real-time computations always take place on time. For a fault tolerant technique detection latency and recovery time are important performance metrics because they contribute to serve down-time. A fault tolerant technique can be useful, in RTDS if its fault detection latency and recovery time are tightly bounded. When this is not feasible, the system must attempt the fault tolerance actions that lead to the least damages to the application’s mission and the system’s users. We have proposed the following scheme that can be used to handle DRB based faults in RTDS. The algorithm makes sure that the backup tasks though scheduled to processors do not hamper the execution of primary tasks at the same time the backup task are updated according to the subtask completed in their primary counterpart so that when the primary task fails the backup task does not start its execution from beginning instead from the last updated subtask. When the primary task is completed within its deadline the backup task is terminated. The global picture of our methodology is shown in Table 1 and Figure 3(a-d).

/* Allocate resources to satisfy task deadline */
Assign the primary and backup tasks to the distributed system in a RANDOMIZED fashion to different processors where the release time of both the primary and backup task is same.
Use the EDF algorithm as uniprocessor scheduling algorithm.
Update the backup task according to the subtask covered by the primary task.
/* Runtime monitoring of timing constraint */
Check if a task misses its intermediate relative deadline with at least M% of the task is completed.
/* Fault Tolerant Strategy*/
If the task misses its deadline then the primary task is terminated and the updated backup task at the scheduled processor is treated as the primary task.
6. If the backup task fails reject the task.

Table 1. High level DRB based Fault Tolerant Algorithm in RTDS.

Algorithm 6.2 mentions our approach for fault tolerance in RTDS and Algorithm 6.1 mentions the fault injection algorithm. The results Figure 4-5 show that our algorithm improves the performance of the traditional Faulty Random-EDF heuristic under permanent fault of 10% and 20% respectively and timing faults of tasks. We present a method that tolerates only permanent and timing failures.

<table>
<thead>
<tr>
<th>T1</th>
<th>Backup</th>
<th>ID</th>
<th>Period</th>
<th>Computation Time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>Deleted</td>
<td>1</td>
<td>3</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>5</td>
<td>32</td>
<td>13</td>
</tr>
<tr>
<td>T3</td>
<td>Primary</td>
<td>3</td>
<td>7</td>
<td>41</td>
<td>2</td>
</tr>
<tr>
<td>T4</td>
<td>Moved</td>
<td>4</td>
<td>11</td>
<td>8</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PID</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
</tr>
<tr>
<td>P2</td>
<td>25</td>
</tr>
</tbody>
</table>

Figure 3(a) Reference of Chart 1, 2.1, 2.2

Algorithm 6.1 Fault Injection Algorithm

1. **Input**: a system resource set G
2. Select a processor $P_i$
3. IF No. of Faults($P_i$) < $N_{fp}$
   Mark $P_i$ as FAULTY
   Increment the Upper Limit.
   [End of if structure]
Algorithm 6.2 Fault Tolerant Scheduling Algorithm

1. Input: a set of periodic task set $T_i = st_1^i, st_2^i, \ldots st_m^i$ and a system resource set $G$

2. Repeat steps from Step 3 to Step 8 for $TIME=1, 2, \ldots$, Sysc-1, Sysc.

3. IF $period(T_k) = TIME$, then:
   INSERT $T_k$ to the Central Scheduler Queue.
   [End of If structure]

4. IF Central Scheduler Queue ≠ NULL
   For each Primary version of task $T_i$
   Select the Non-Faulty end node $P_1$.
   INSERT the Primary task to the Primary Queue ($P_1$)
   Create a Backup version of task $T_i$
   Select the Non-Faulty end node $P_2 \neq P_1$
   INSERT the Backup task to the Backup Queue ($P_2$)
   [End of If structure]

5. IF $period(Fault Injection Algorithm (System Resource Set)) = TIME$
   FAULT INJECTION ALGORITHM (System Resource Set)
   [End of If structure]

6. For each processor $P_i$ in the system
   a) IF new task added to the Primary Queue
      Rearrange the tasks in the Primary Queue of $P_i$ according to EDF.
      IF $Deadline(Primary Queue [Front]) > Deadline(Primary Queue)$
      Preempt the currently executing task.
      Assign Primary Queue [Front] to the processor.
      [End of If structure]
   [End of If structure]
   b) Execute the task $T_i$ assigned to Processor from the Primary Queue.
   c) IF Intermediate Deadline ($T_i$) exceeds TIME
      Terminate the primary version of task $T_i$.
      Trigger a timing fault alarm.
      Intimate the Backup version of task $T_i$ on remote processor $P_i$.
      [End of If structure]

7. Update backup version of $T_i$ to the last valid subtask of the primary version of $T_i$.

8. [Update the task from Backup Queue to Primary Queue if the primary task has failed]
   a) Rearrange tasks in the Backup Queue whose primary task has failed according to EDF.
   b) DELETE Backup Queue [Front] and INSERT in Primary Queue.
   c) Rearrange all the tasks in the Primary Queue according to EDF.
7. Experimental Analysis

To evaluate how well the proposed scheme performs, we compare the performance of *FT Random-EDF* with *Faulty Random-EDF* using a discrete event simulator developed by us using Matlab 6.0. The tasks are arriving into the systems dynamically in a periodic fashion, which are assigned to the processor by random selection, provided the processors memory is not full. The uniprocessor scheduler used is EDF algorithm. Timing fault are injected into the system when a task misses its deadline whereas crash or permanent fault are injected in random fashion with an upper an upper limit of 10% and 20% respectively. A suitable fault monitoring technique is used to detect the timing fault. Experiments have been conducted by varying the computational requirements of the periodic task. Our algorithm has been used at the end-node to try and improve the overall systems performance in terms of throughput as the performance metric.

The results Figure 4-5 show that our algorithm outperforms the traditional EDF uniprocessor scheduler, which has missed deadline in presence of timing and crash fault, and a randomized assignment of tasks.

8. Conclusion

Fault-tolerance then becomes an important key to establish dependability in these systems. Hardware and software redundancy are well-known effective methods for hardware fault-tolerance, where extra hard ware (e.g., processors, communication links) and software (e.g., tasks, messages) are added into the system to deal with faults. We have investigated methods to overcome timing and permanent failures in heterogeneous real time distributed systems with point-to-point communication links. We have proposed a new method that tolerates at most NFP permanent fault and arbitrary timing fault. This method is a software solution based on adaptive redundancy to overcome the failures. The percentage increase in the throughput of FT Random-EDF than Faulty Random-EDF is 76% with upto 10% of faulty systems for processor range in between 3 to 50.

9. Acknowledgement

The work reported in this paper are being supp orted in part by R&D project grant 2005-2008 of MHRD Government of India with the title as “Fault Tolerant Real Time Dynamic Scheduling Algorithm For Heterogeneous Distributed System” and being carried out at department of Computer Science and Engineering, NIT Rourkela.

---

References


