Fault Tolerance

This chapter presents fault tolerance and resource utilization. Fault with respect to software systems is the inability of hosts not to communicate with other hosts due to hardware or system failure in extremely loaded situation. Fault tolerance is to overcome these types of situations. A solution to fault tolerance is checkpointing. A checkpoint is a snapshot of the current state of a process. It saves enough information on non volatile stable storage such that, if the contents of the volatile storage are lost due to process failure, one can reconstruct the process state from the information saved on the non-volatile stable storage. Rest of the chapter is organized as follows:

- Section 8.1 gives issues regarding fault tolerance and resource utilization
- Section 8.2 describes fault tolerance and its components FM and FD
- Section 8.3 gives Problem formulation and assumptions
- Section 8.4 describes MACP algorithm
- Section 8.5 gives analysis
- Section 8.6 presents optimal resource utilization
- Section 8.7 describes the dominating factors in optimal resource utilization
- Section 8.8 describes GA
- Section 8.9 gives result and discussion
- Section 8.10 summarizes the chapter.

8.1 Issues

Conventional methods for fault tolerance use message passing for checkpointing with rollback which is expensive mechanism because multiple
local processes need to participate in sessions of message passing. Furthermore, existing algorithms are mostly designed for closely coupled distributed systems. They may not scale to the world-wide Internet applications because it may incur large overhead w.r.t. network traffic and message passing delay. The key issues in fault tolerance are fault management, optimal resource utilization, network delay and to recover from the fault quickly.

8.2 Problem Formulation and Assumptions

To address the issues mentioned in Section 8.1, we have developed MA assisted checkpointing algorithm. We consider a distributed system consisting of \( n \) processes, one process per processor, denoted by \( \{ P_0, P_1, \ldots, P_{n-1} \} \). There is no common clock, shared memory, or central coordinator. Any process can initiate checkpointing process. We assume that there is no link failure, only processes may fail. The computation is asynchronous; messages are exchanged with finite but arbitrary delay. We consider logical checkpoint, which is a standard checkpoint (i.e., snapshot of the process) plus a list of messages, which have been sent by the respective process but are unacknowledged at the time of taking the checkpoints. Message lists are updated continuously. For each process, at most two checkpoints may have to be stored on the stable storage when checkpointing procedure is running.

In a distributed system, each process has to take checkpoints periodically on non-volatile stable storage. In case of failure, the system rolls back to the consistent set of checkpoints. If all the processes take checkpoints at the same time, the set of checkpoints would be consistent. But since globally synchronized clocks are very difficult to implement, processes take checkpoints within an interval. They are made permanent, when all processes agree.
Basically, checkpointing algorithms are classified into two broad categories: Synchronous and Asynchronous. In asynchronous checkpointing each process takes checkpoints independently, without bothering about the other processes. In case of a failure, after recovery, a Consistent Global Checkpointing State (CGS) is found among the existing checkpoints and the system restarts from there. In synchronous checkpointing all processes have to synchronize through control messages before taking checkpoints. These synchronization messages contribute an extra overhead.

8.3 Fault Tolerance

To describe the solution of fault tolerance, we have designed and implemented a MA based prototype consists of Fault Detector (FD) and a Fault Manager (FM). FD is responsible for monitoring the state of resources and detects the resource failure. FM is responsible for resolving the detected failures. FM requests FD to monitor the state of a resource and FD periodically reports resource state information to FM. Processes are MAs enabled and activate the respective MA as soon as they want to take a checkpoint.

8.3.1 Fault Detector (FD)

FD detects an occurrence of failure by analyzing the information about the state of a resource and transfers this information to FM. If FM receives the information about the failure, it tries to resolve that failure. FD provides three types of services- monitoring, decision and communication. A monitoring service monitors the state of processes, processors, and network. A fault detection service decides the failure occurrence for each resource. A communication service provides communication within each component. For a fault detection service, FD consists of a process monitor agent, a processor monitor agent, a network monitor agent, and fault decision agent.
8.3.2 Fault Manager (FM)
FM requests FD to monitor the state of each resource and FD periodically reports resource state information to FM. FM is responsible for resolving detected failures. FM also provides a display service that shows each resource state to user, a task migration service for failure resolution and a communication service that provides communication with each component. For a fault management service, FM consists of a state display agent, a task migration agent, and rescheduling agent.

8.4 Algorithm
We have implemented Mobile Agent assisted Checkpointing (MACP) algorithm for fault tolerance. In this algorithm, for each process, at most two checkpoints which are stored on the stable storage have been taken. Initially in Step1, these checkpoints are temporary. In Step 2, these checkpoints are made permanent.

Checkpoints are assigned a unique number $chk_{no}$. In the beginning all processes start by taking checkpoints with $chk_{no}=0$. The algorithm is non-blocking, i.e., even when a checkpointing process is running, processes are free to run their applications. Each process has a list of all its neighbors and initiate checkpointing independently. The initiator creates a MA, which travels across the network to create CGS. The MA id is the same as the process id of its creator.

MAs move to other processes following an execution path of depth-first search, starting from its creator. The agent maintains a stack and two lists. The stack, $Stack$, has the path to the root. The list $VL$ (Visited List) is the list of processes which have been visited by this agent before any other agent in this checkpointing cycle. $PL$ (Partial List of concurrent Initiators) is a list of processes who have initiated check pointing.
The algorithm allows the generation of missing messages in case the system has to roll back to its last checkpoint. At the time of restart after a failure, processes retransmit their unacknowledged messages. So there may be duplicate messages after recovery from a failure and they are handled using message identifiers.

In MACP, we have used MAs to enhance the design of distributed rollback recovery algorithm. MAs act as messengers and/or monitors travel over the network from site to site and facilitate the coordination of the distributed computation processes to carry out their check pointing and rollback activities.

**Step 1:** This step is executed by FD. We are only considering the process failure and use generalized term agent for all the agents as a part of detection. Suppose process $P_i$ is a checkpointing initiator. $P_i$ creates a MA with agent id = $i$. The agent takes a temporary checkpoint $\text{chk\_state}=T$ and $\text{chk\_no}$ equal to $\text{chk\_no}+1 \mod 2$. We have assumed that at the most two checkpoints are permissible for each process- temporary and permanent. The process id of the current process $i$ is pushed onto the stack $S_j$. Then the agent leaves $P_i$ and moves to the neighbor of $P_i$ say $P_j$. When an agent, $MA_i$ visits process $P_j$, it halts the application process of $P_j$, adds its agent id to $PL_j$. If it finds that no checkpoint has been taken on the process $P_j$, $MA_i$ takes a new temporary checkpoint for $P_j$ with $\text{chk\_state}=T$ and $\text{chk\_no} = (\text{chk\_no}+1 \mod 2)$. If any neighbor of $P_j$ is yet to be visited, the agent moves to that neighbor and performs the same operation. Before leaving $P_j$, $MA_i$ pushes $j$ onto stack $S_j$. If all neighbors of $P_j$ have already been visited, then $MA_i$ moves back to the last process from which it came to $P_j$. The id of the last visited process is available at the top of stack $S_j$. Before leaving $P_j$, $MA_i$ puts $j$ in $VL_i$. If $P_j$ has
already taken a temporary checkpoint (i.e., \( \text{chk\_state} = T \)), i.e., another agent, from a different initiator, has already visited this process.

**Step 1: MA on being initiated by the process takes temporary checkpoints**

1. Process \( P_i \) creates a agent \( MA_i \) with \( id \leftarrow i. \)/* Enable Process Monitor Agent */
2. Push \( i \) on to the stack \( S_i. \)
3. \( \text{take\_temporary\_checkpont} (i, i) /* Taking temporary checkpoint on process \( i \) by \( MA_i */
4. While \( (S_i \neq \text{null}) \)
5. \( (\text{top\_S}_i \neq \text{null}) \) for neighboring host and it is not visited by \( MA_i, \) Move to \( P_j. \)
6. \( \text{take\_temporary\_checkpont} (i, j) /* Taking temporary checkpoint on process \( j \) by \( MA_i */
7. Take a new checkpoint and update the state as \( \text{chk\_state} = T \).
8. \( \text{chk\_no} = \text{chk\_no} + 1. \)
9. \( PL_j = PL_j \cup i^j \)
10. \( VL_i = VL_i \cup A^j \)
11. if process \( P_j \) has taken a temporary checkpoint \( T, \) \( PL_i = PL_i \cup PL_j \) and set \( PL_j = PL_i \)
12. Else Agent \( MA_i \) returns back to process \( P_i \)
13. If \( P_j \) is unvisited by any other agent, Push \( j \) on to stack \( S_i. \)
    Else Pop stack \( S_i \) and move to Top of stack \( S_i. \)
14. End while
15. Set the \( \text{first\_step\_flag} = \text{true} /* Enable Fault Decision Agent */

**Figure 8.1:** FD Algorithm

So the information about the list of initiators is available with the agent in \( PL_i \) and the combined information replaces both \( PL_i \) and \( PL_j \).
Finally, the agent leaves $P_j$ and moves to unvisited neighbor of $P_j$. At the end of first step, agent $MA_i$ returns back to $P_i$. In this algorithm we have considered only process failure, so accordingly the corresponding agent is chosen.

**Lemma 8.1:** At the end of first step, every process has one temporary checkpoint with same $chk_{\_no}$ and this set is consistent. (Proof of this lemma is given in Appendix C)

**Lemma 8.2:** In the first step, if the number of concurrent initiations is $k$, the total number of moves by all the agents is $O(\text{kn})$. (Proof of this lemma is given in Appendix C)

**Step 2:** The temporary checkpoints taken in first step are made permanent in the second step. The complete list of initiators is communicated to all other initiators by the agent initiator. Suppose process $P_i$ is the agent initiator which is selected from $(MA_1, MA_2, \ldots, MA_n)$. It creates two different agents. The first one visits the processes in $VL_i$ and confirms the temporary checkpoints, deleting the old permanent checkpoints. The second agent visits the processes in $PL_i$. $P_j \in PL_i$ when visited by the second agent, activates $MA_j$, which confirms the temporary checkpoints for processes in $VL_j$. Finally all agents return to their creators and are destroyed.

**Lemma 8.3:** In the second step, the total number of moves for the agents is $O(n)$.

(Proof of this lemma is given in Appendix C)
Step 2: Temporary checkpoints taken in Step 1 are made permanent

1. \( MA_i \) at \( P_i \) /* Enable State Display Agent */

2. Push \( i \) onto stack \( S_i \).

3. \( MA_i \) moves to \( P_j \) such that \( j \in PL_i \).

4. Create a clone of \( MA_i \).

5. Move clone to \( P_k \) such that \( k \in VL_i \).

6. \( MA_i \) at \( P_j \) where \( j \in PL_i \) /* Enable Rescheduling Agent */

7. Push \( j \) onto stack \( S_j \).

8. While (stack \( S_i \neq null \))

9. Recreate \( MA_j \) if it is destroyed.

10. Move \( MA_j \) to \( P_k \) such that \( k \in VL_j \).

11. if \( k \in PL_i \) is unvisited Move to \( P_k \).

12. Else Pop stack \( S_j \) and move to top of stack \( S_i \).

13. End While

14. \( MA_i \) at \( P_j \) where \( j \in VL_i \)

15. Push \( j \) into stack \( S_i \).

16. While (stack \( S_i \neq null \))

17. Delete old permanent checkpoints.

18. \( \text{chk\_state} \leftarrow P \) /* Temporary checkpoints \( \leftarrow \) are made permanent*/

19. if \( k \in VL_i \) is unvisited Move to \( P_k \).

20. Else Pop stack \( S_j \) and move to top of stack.

21. End while

Figure 8.2: FM Algorithm
**Theorem 8.1:** At the end of second step, every process has one permanent checkpoint with same \( \text{chk\_no} \) and this set establishes a CGS of the system.

(Proof of this Theorem is given in Appendix C)

**8.4.1 Analysis**

To compare the designed algorithm message complexity of the designed algorithm is compared with Parkash and Singhal (P-S) [66], Spezialetti and Kearns (S-K) [69]. When the designed algorithm is executed by MA the overhead in terms of message complexity is very less compared to traditional Message Passing based checkpointing and rollback algorithms. In the case of the designed algorithm, worst case message complexity is \( O(n^2) \) compared to earlier proposed algorithms, Parkash and Singhal (P-S) [66], Spezialetti and Kearns (S-K) [69], which have worst case message complexity as \( O(n^3) \). As given in Table 8.1 (Appendix B), worst case message complexity of designed algorithm is \( O(n^2) \).

**8.5 Optimal Resource Utilization**

A resource is a reusable entity that is used to fulfill a task or user request. It could be a host, network, or some other service that is synthesized using a combination of hosts, networks, and software. The resource provider is an agent that controls the resources. Similarly, a resource consumer is an agent that controls the consumer resources. Due to issues such as extensibility, adaptability, site autonomy, QoS, and co-allocation, resource management is challenging task in distributed computing environments (DCEs). In DCEs, due to resource heterogeneity and security concerns it is better to execute a task only on subset of resources. Therefore, optimal resource allocation is implemented in such systems, to a certain extent using task priorities. It is essential for the RM to consider the tasks access privileges, type of subscription, and resource requirements while determining the level of
optimal resource allocation. Depending on the type of optimal resource allocation, a contract is formed between RM and tasks. When a task overruns its resource usage predictions, RM would ensure that it does not infringe on the resource allocation of other tasks, i.e., the RM provides isolation among the tasks for fair resource management.

8.6 Factors effecting optimal Resource Utilization

Today, resource utilization and task migration are most common issues in dynamic load balancing algorithms. In a situation where newly created tasks arrive randomly into the system, some of the processors become heavily loaded while others are idle or lightly loaded. Therefore, the main objective of load balancing is to develop task assignment algorithms to transfer or to migrate tasks from heavily to lightly loaded processors so that no processors are idle while there are other tasks waiting to be processed. In this way, an effective resource management is achieved. In general, a dynamic load balancing algorithm consists of three major issues- the load measurement, information exchange and initiation, and load balancing operation.

Load Measurement

Load information is typically quantifiable by a load index value set to zero when the processor is not loaded and increases, as the processor load gets heavier. As load measurement occurs frequently, the calculations must be very efficient, which rules out any exhaustive use of too many parameters. Simple parameters, such as the response time or completion time (of all tasks), are sufficient in most of the cases.

Information Exchange and Initiation

Information exchange specifies how to collect and maintain the network load information necessary for making load balancing decisions. This is a balancing act between cost of collecting global load information and maintaining the accurate state of the system. There are three basic
information exchange rules. In periodical information exchange, individual processors will report the load information to each other periodically at a predetermined time interval. This exchange rule is good in the sense that the load balancing operation can be initiated based on the maintained workload information without any delay. However, the main problem lies in setting up the interval for information exchange. An interval that is too long causes an inaccuracy in decision making while a short time interval incur heavy communication overhead. Hence, a centralized rule is used in which a dedicated processor will collect and maintain the system's load information. New load balancing decisions are made based on current workload information as well as feedback from previous decisions. Moreover, the initiation rule is used to decide when to begin load balancing operations.

**Load Balancing Operation**

The load balancing process can be defined by three rules: the location rule, the distribution rule, and the selection rule. The location rule determines which processors are involved in the balancing operation. Load balancing domains can be either global or local. A global domain allows the balancing operation to transfer load from one processor to any of the processors in the system, while a local domain only allows balancing operations to be performed within the set of nearest-neighbor processors. The distribution rule determines how to re-distribute the workload among processors in the balancing domain. This rule depends on the balancing domain that is determined by the location rule, while the selection rule decides on whether the load balancing operation is performed preemptively or non preemptively.

**8.7 Genetic Algorithm (GA)**

To deal with all the issues defined above, we have developed GA for optimal resource allocation. GA is a search algorithm based on the principles of evolution and natural genetics. GA combines the exploitation of past results
with the exploration of new areas of the search space. By using survival of the fittest techniques combined with the structured, yet randomized information exchange, a GA optimizes the solution of a given problem. A generation is a collection of artificial creatures (strings). In every new generation, a set of strings is created using information from the previous ones. Occasionally, a new part is tried for good measure. GA is randomized, but they are not simple random walks. It efficiently exploits historical information to speculate on new search points with expected improvements [175-178]. The majority of optimization methods move from a single point in the decision space to the next using some transition rule to determine the next point. This point-to-point method is dangerous as it can locate false peaks from the available many peaks. By contrast, GA forms a database of points simultaneously (a population of strings), climbing many peaks in parallel. The probability of finding a false peak is reduced compared to methods that move point-to-point. The mechanism of a simple GA is simple, involving nothing more complex than copying strings and swapping partial strings. Simplicity of operation and power of effect are two main attractions of the GA approach. The effectiveness of the GA depends upon an appropriate mix of exploration and exploitation. Following three operations are key in GA: selection, crossover, and mutation. Selection according to fitness is the source of exploitation. The mutation and crossover operators are the sources of exploration. The trade-off between exploration and exploitation is clearest with mutation. As the mutation rate increases, mutation becomes more disruptive until the exploitative effects of selection are completely overwhelmed. More information is provided on these operators in [175]. Following are the steps of GA for optimal resource allocation:
Step 1: A fixed number of tasks, each having a task number and size, is randomly generated and placed in a task pool from which tasks are assigned to processors. As load-balancing is performed by the centralized GA based method, the first thing to do is to initialize a population of possible solutions [179]. This can be achieved by using sliding window technique. Since the objective is to determine task schedules on the fly as the tasks arrive, the strings in the population are used to represent the list of tasks known to the system at that time. However, as there may be too many tasks waiting to be assigned at a time, the sliding-window technique is used so that only tasks that are within the window are considered for execution each time [180]. Window size is fixed, with the number of elements in each string equal to the size of the window. Then, permutations of these tasks is performed from the lists that represent the different orders in which these tasks are scheduled for execution. When GA arrives at a task schedule, these tasks will be assigned to processors accordingly. Once these tasks have been assigned, the sliding window will be updated with new tasks by sliding along to the next set of tasks on the task queue and repeating the assignment process.

Step 2: Objective and Fitness Functions

The objective function is the most important component of any optimization method, including GA. It is used to evaluate the quality of the solutions. The main objective here is to make task assignments that will achieve minimum execution time, maximum processor utilization, and a well balanced load across all the processors. So the objective function is incorporated into the fitness function of GA. This fitness function is used to measure the performance of the strings with respect to the objectives of the algorithm.

2.1 Calculation of Largest Task Completion Time

The first objective function for the GA algorithm is the effective task schedule [177], which is represented by a string. It is basically the largest
task completion time among all the processors in the system. However, as the processors may not always be idle when the string is evaluated, it is inaccurate to calculate task completion time by considering the sizes of these new tasks alone. The current existing load on each individual processor is also taken into account in order to get a more accurate task completion time value.

\[ P(\text{Completion time}) = (\text{Current load of } P) + \text{new load assigned to } P \]

### 2.2 Average Utilization

The next factor to the fitness of a string (task schedule) is the average processor utilization. This is essential since high average processor utilization indicates that the load is well balanced across all processors. By keeping the processors highly utilized, the total execution time is reduced.

The expected utilization of each processor based on the given task assignment is calculated. This is achieved by dividing the task completion time of each processor by the task completion time value. The utilization of the individual processors is given by-

\[ P(\text{Utilization}) = \frac{P(\text{Completion time})}{\text{task completion time}} \]

### 2.3 Number of Acceptable Processor Queues

The overall task assignments being evaluated have a small task completion time and high average processor utilization. However, assigning these tasks to the processors may still overload some of the processors. Therefore, the third objective is to optimize the number of acceptable processor queues. Each processor queue is checked individually to see if assigning all the tasks on the processor queues will overload or underload the processors.

Whether a processor queue is acceptable or not is determined by the light and heavy thresholds used. If the task completion time of a processor (by adding the current system load and those contributed by the new tasks) is
within the light and heavy thresholds, the processor queue is acceptable. If it is above the heavy threshold or below the light-threshold, then it is unacceptable. In order to optimize this objective, the percentage of acceptable processor queues is calculated. This is achieved by dividing the number of acceptable processor queues by the total number of processors in the system. Higher the percentage, better the schedule is in terms of its load balancing potential.

The three objectives discussed above are incorporated into a single fitness function and given by the following equation:

\[
\text{fitness} = \frac{1}{\text{task completion time}} \times \frac{\text{avg. utilization}}{\text{avg. utilization}} \times \frac{\text{queues}}{\text{processors}}
\]

**Step 3: Selection, Crossover, and Mutation**

The selection technique is used based on the roulette wheel method [175]. In this case, the slots of the roulette wheel are determined, based on the probabilities of individual strings surviving into next generation. These probabilities are calculated by dividing the fitness values of the individual string by the sum of the fitness values of the current pool of strings. Adding the probability of current string to the probability of the previous string creates the slots. For example, if the probabilities of surviving for string 1 and 2 are 0.25 and 0.3, respectively, then slot 1 will range from 0-0.25 while slot 2 will range from 0.26-0.55. The slots are allocated up to the value of 1. Hence, each string in the population is to occupy a slot size that is proportional to its fitness value. After defining these slots, random numbers between zero and one are generated. The numbers obtained will determine which strings will survive into the next generation. As the fitter strings are represented by larger slots on the wheel, the chances of the randomly generated numbers falling into the slots that represent these strings are
higher. After completing the selection process, the fitter strings left in the pool. Then, the crossover operation is performed on pairs of strings that are picked at random from the population. However, the normally used single point crossover method is not applied to this case, as using this method may cause some tasks to be assigned more than once while some are not assigned at all. Therefore, in order to ensure that each task is only assigned once, the cyclic crossover method is adopted [181]. With this crossover method, the recombination is performed under the constraint that each task comes from one parent or the other.

**Step 4: Evaluation of Strings**

After completing the mutation process, the populations of strings are evaluated using the fitness function. Each individual string has a new fitness value and a new probability of surviving into the next generation. These values are used to define the slots of the roulette wheel in the next GA cycle. The total and average fitness values of the strings in the pool are calculated after every cycle to compute if new pool of strings is fitter than the ones in the old pool. Instead of waiting for the GA to converge, it is allowed to run for a fixed number of k cycles. The decision is made because solutions generated in less than k generations may not be good enough. On the other hand, running the GA for more than k generations is not be very feasible, as too much time is devoted to GA operation.

When the GA is terminated after k cycles, the fittest string in the pool survives and used as the task schedule. As the GA is designed for the purpose of load balancing, it is important that the task assignments do not result in any idle processors. Hence, if the fittest assignment determined by GA results in any processors being idle, the assignment will be rejected.
8.8 Result and Discussion

The performance of the designed algorithm is compared with the First Fit (FF) algorithm [182] and a random allocation scheme [183]. However, as random allocation does not provide consistent results that can be used for meaningful analysis, it is more important to focus on the results of GA and the FF algorithm in most of the cases.

Initially, the test runs are based on the set of default values: number of tasks = 100, number of processors = 5, window size = 10, number of generation cycles = 5, population size = 10, maximum size of each task = 20. The measurement of performance of these algorithms is based on two metrics: total completion time and average processor utilization. The default parameters are varied and the results are collected from test runs which are used to study the effects of changing these parameters. Only one parameter is changed each time so that any changes in performance are based solely on this parameter.

8.8.1 Changing the Number of Tasks

Default values are used for all the parameters except for the number of tasks to be assigned. The numbers of tasks are varied from 20-1000 and the effect on the total completion time and average processor utilization is given below:

Completion Time

Figure 8.3 presents the total time for all three algorithms. It increased linearly as the number of tasks increased. This is expected, as the more tasks to be scheduled, it takes longer to complete all the tasks. It is also observed that the GA performed better than the random allocation and the FF algorithm in all cases. When comparing the results of the GA and the FF algorithm, we have observed that the gap between these two curves is widened as the number of tasks increased. This shows that the GA actually
reduced the total completion time by considerable amount in comparison to the FF algorithm as the number of tasks increased. This also indicates reliable performance of the GA when the number of tasks increased.

**Average Processor Utilization**

Figure 8.4 presents that GA outperformed FF algorithm in terms of processor utilization. The processor utilization using the GA and FF algorithms are 85-99 % and 80-90 % respectively. It is also noticed that the processor utilization of the GA actually approached to 99 % as the number of tasks are increased, which indicates that a GA works better in case of more tasks. This is quite consistent with other results in the literature [183].

![Figure 8.3: Total Completion Time](image-url)
8.8.2 Varying the Number of Processors

In this section, the size of the window and the number of processors are varied to observe their effect on the GA performance. It is important to note that the numbers of processors are changed according to the size of the window because an inappropriate number of processors used may prevent the GA to work properly. A window size that is too small for number of processors available may stop the algorithm from working accurately. It is mentioned earlier that if the fittest task assignment results in any processor being idle, it will be rejected and the GA is initiated again to generate another assignment. Therefore, when there is a situation where there are 25 processors and the size of the window is 10, the chances of having at least one task on each processor is nil. GA will be re-executed over and over again, but a task mapping that works for this load balancing mechanism will never be generated. Hence, it is crucial that the window size correlates with
the number of processors in the system. The effects of changing the window size (and the number of processors) from 10 to 50 are shown in Figures 8.5 and 8.6.

Completion Time

Figure 8.5 presents that the completion time improved when the window size is increased. However, this improvement is mainly caused by the fact that more processors are used for larger windows. Therefore, even though there are more tasks to be scheduled in each round, the extra load is easily handled by the additional processors.

![Figure 8.5: Total Completion Time](image)

Average Processor Utilization

Unlike the performance improvement in terms of total completion time, Figure 8.6 presents the average processor utilization deteriorated as the size of the window increased. This shows that load balancing is harder when more tasks are to be balanced across a larger system. Hence, improving the
processing time by increasing the window size and number of processors comes at the cost of lower processor utilization.

Figure 8.6: Average Processor Utilization

8.8.3 Varying the Number of Generation Cycles

The number of generation cycles for the GA is changed to observe the effects of this parameter on the performance of the algorithm.

Completion Time

Figure 8.7 shows that the total completion time is significantly reduced as the number of generations are increased from 5 to 20. This is due to the fact that the quality of the generated task assignment improves after each generation. However, after 20 generations, it is observed that running the GA does not seem to improve performance much. The results suggest that running the GA for more generations will be superfluous, as the improvement will only be marginal.
Average Processor Utilization

Besides reducing the total completion time, the average processor utilization also improved as GA is executed for more generations (Figure 8.8). However, once again, the improvement achieved is slowed down as the solution converged. Every test run (for a different number of generations) is based on different sets of tasks. Running for more generations does not mean that the same population of solution is being operated. Therefore, the results may be slightly different if the same set of solutions is being optimized.

![Figure 8.7: Total Completion Time](image)

Figure 8.7: Total Completion Time
Figure 8.8: Average Processor Utilization

8.8.4 Changing the Population Size

Different population sizes are used to generate the results. The population sizes ranged from 5 to 40. The effect of changing population on chosen metrics is as follows:

**Completion Time**

It is observed that increasing the population size does not increase the performance too much. Even though an improvement is shown in Figure 8.9, but this improvement is only minimal. Despite that, choosing an appropriate population size is still very important as one that is too small causes the GA to converge too quickly, while one that is too large results in long waiting time for significant improvement [184]. Therefore, if the population pool is larger, the computations involved could be more intensive, hence resulting in high processing costs. So, using a population size that is too large could not provide optimized solution [185].
Average Processor Utilization

Despite minimal improvement in terms of total completion time, increasing the population size had a positive effect on processor utilization. The processors are better utilized since a larger search space gives GA a better opportunity to find better and fitter task mappings (Figure 8.10).

![Figure 8.9: Total Completion Time](image-url)
8.9 Summary

In this chapter, we have evaluated a prototype for fault tolerance consists of FD and FM. FD detects the type of fault and FM provides the diagnosis of the fault. The process of diagnosis is MA enabled. An algorithm called MACP is developed for this purpose. The complexity of the algorithm is better than the other existing algorithm for the same purpose. Also a GA is developed for optimal resource selection and allocation. GA based scheme works better when the numbers of tasks are large. GA worked well in terms of achieving the goals of minimum total completion time and maximum processor utilization. Hence the designed prototype works well to achieve the dual goal of fault tolerance and resource utilization.

In the next chapter, we will present security and intrusion detection in MANET.