Journal of Mobile Multimedia, Vol. 5, No. 2 (2009) 113–124
© Rinton Press

FAULT TOLERANCE IN THE MOBILE ENVIRONMENT

DANIEL C. DOOLAN
School of Computing, Robert Gordon University
Aberdeen, AB25 1HG, United Kingdom
d.c.doolan@rgu.ac.uk

SABIN TABIRCA
Department of Computer Science, University College Cork
Cork, Ireland
s.tabirca@cs.ucc.ie

LAURENCE T. YANG
Department of Computer Science, St. Francis Xavier University
Antigonish, NS B2G 2W5, Canada
lyang@stfx.ca

Received April 2, 2008
Revised January 20, 2009

In general it is assumed that a parallel program will execute on reliable hardware. A fault tolerant program and underlying infrastructure should be capable of surviving failures such as system crashes and network failures. At the highest level the application should be capable of automatically recovering from a set of faults without any change to the apparent behaviour of the program. The process of checkpointing may be used to allow a program to save its state to persistent storage, abort and restart from the checkpoint. Several fault tolerant MPI implementations are currently in existence, for example MPICH-V is considered to be one of the most complete, featuring checkpointing and message logs to allow aborted processes to be replaced. No matter how sophisticated a fault tolerant system may be, it can never be completely relied upon, as there is always the possibility of a complete system failure. It is one thing to develop fault tolerant applications on high end dedicated clusters and supercomputers, however applying fault tolerance to the realm of mobile parallel computing presents an entire new series of challenges that are inexorably linked with the unpredictable nature of wireless communication systems. Two differing strategies for fault tolerance in the mobile Bluetooth wireless environment will be presented and compared to see which should be adopted over another.

Keywords: Fault Tolerance, Mobile Message Passing, Bluetooth

Communicated by: D. Taniar & I. Khalil

1 Introduction

The Mobile Message Passing Interface (MMPI) [4] is a library built upon the principles of MPI and relies on Bluetooth technology to provide the underlying wireless infrastructure for inter-node communications. Unlike standard Bluetooth networks that are of a Star network topology (in the case of a Piconet). The MMPI system uses a fully interconnect mesh network to facilitate intercommunication between each individual node within the network. Thus, just as with MPI one can communicate in either a point to point fashion or globally by using a series of simple communications method calls.

The creation of the world is quite a lengthy process due to the times involved for device and service discovery. According to the Bluetooth specification the inquiry phase must last for 10.24 seconds [2], this can take quite a bit longer when executed on a physical device. The process of service discovery can take just as long in the case that several devices are found. Finally once all the nodes that are advertising the "mmpiNode" service are detected, the process of forming the mesh can take place. This can take as little as one to two seconds depending on the number of nodes. In total one can expect to wait on the order of twenty seconds for the network formation process to complete.

The library has proven to be very effective in many application domains from parallel computing and graphics to mLearning and multiplayer games. It is especially useful in the area of gaming as a well designed single player game can be transformed into a multi-player game with minimal code changes. One should always anticipate that errors in communications will occur, especially in the case of a wireless system. Therefore the two strategies for fault tolerance presented are with respect to this new library for mobile parallel computing.

1.1 Fault Tolerance in MMPI

As previously mentioned a truly fault tolerant system can never be achieved within an MPI implementation. This is certainly more so true, when nodes are connected over wireless communication media, as this adds a whole new dimension of possible faults into the system. To add fault tolerance capabilities to the MMPI system several distinct mechanisms have been included into the implementation. Checkpointing by the writing of system state to persistent storage in the form of the Record Management System (RMS), and system reconfiguration in the form of shrinking or rebuilding the communicator in the event of a failure of a node.

The reconfiguration of the communicator may be suitable for some application domains but not for others. In the case of mobile gaming for example, the reconfiguration in the case of shrinking the communicator in the event of the loss of a node, would require minimal time to carry out the shrinking process. The avatars of the game itself would have to be updated to reflect the removal of one from the game. In the case of a parallel computation the shrinking of the communicator may not be an option in the event that global communication methods such as scatter and gather were used. The data that each node would contain would be reliant on the overall size of the communicator itself, and thus the data for processing would have been evenly distributed over a certain number of nodes in the system. Checkpointing would therefore be an essential element in such systems, as well as the rebuilding of the communicator to ensure the communicator size remains consistent.

1.2 System Architecture

The addition of fault tolerance to MMPI required the addition of several more classes for handling the checkpointing mechanism and persistent storage operations (Figure 1). System reconfiguration and error notification required updates to the main MMPI, BTServer and BTClient classes. Due to the fact that JSR-82 Bluetooth does not allow low level access to the HCI layer for the detection of errors, the process is carried out through the use of IO Exception handling. The CheckPT class carries out the process of checking the worlds state and initiating the saving and restoration of same. The StateHandler class is use by CheckPT for the reading and writing of state information to persistent storage, be it an to File or RMS database.

The MMPI class is the main interface through which a developer may access the library. The creation of an instance of this class will instigate the formation of a parallel world. The MMPI object of each node maintains an array of DataStreams both for input and output that are connected to every other node in the world. A node may retrieve its particular rank within the world and the size my using the methods getRank() and getSize(). A job may then be distributed throughout the nodes of the system based on these values. To achieve internode communication one may simply call methods of the instantiated object. Methods such as send(...) / recv(...) allow for point to point data transfer, while methods such as bcast(...) and scatter(...) allow for global communication.

The MMPI class is supported by two further classes, namely BTClient and BTServer. When the constructor of the MMPI class is called, a parameter is passed to designate whether is should be activated in either Client or Server mode. This in turn will create an instance of either the BTClient or the BTServer. These classes allow for the creation of the appropriate underlying Client / Server architecture necessary for Server registration and Discovery.

Should a developer wish to produce parallel graphics applications they may make use of the additional classes MMPE and MMPECanvas. The MMPE class provides and interface to the developer for the drawing of graphic primitives. These can be drawn as with MPI on the root node of the system, but a broadcast equivalent of each method also exists, allowing for the graphics to be rendered on all of the devices within the parallel world.

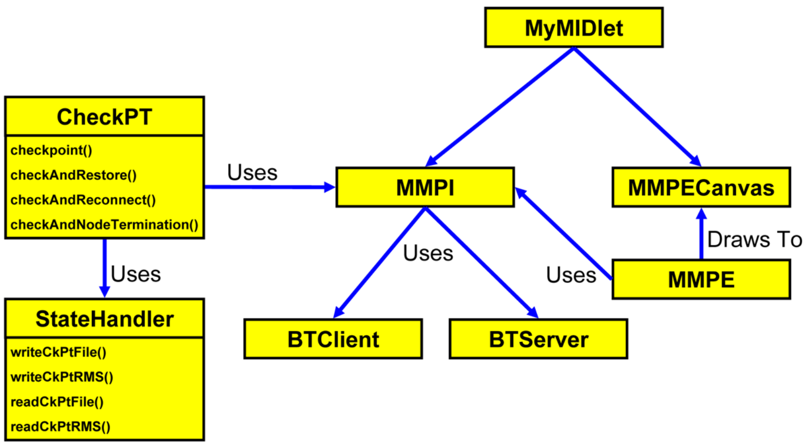

Fig. 1. MIDlet Structure with Fault Tolerance Classes.

1.3 Checkpointing

Checkpointing is a procedure whereby state information is continually saved to persistent storage to allow restarting of the computation from the checkpoint in the case of a system failure. The creation of a checkpoint is often considered to be an expensive operation as it involves slow disk I/O. This requires system time that could otherwise be allocated to the additional computation of the task at hand. Factors that can impact on the cost of checkpointing include: the cost of creation and writing, the cost to read and restore, the probability of failure, and the time between checkpoints. Clearly the practically of checkpointing is directly related to the performance of persistent IO operations.

Two possibilities exist for the creation of checkpoints, that of user-directed and system-directed checkpointing. In user-directed implementations the programmer is responsible for creating the checkpoints. Two drawbacks exist with this system, the developer is responsible for ensuring all necessary state data is saved. Secondly checkpointing should occur at points within the application where no inter-device communication is necessary. Achieving this can be difficult for a program that does not have a distinctive iterative structure. One could also develop for the creation of checkpoints only when a failure is detected. Some work has been done in regard to system-directed checkpointing, but the extraction of all the necessary state can prove difficult, and the state may include time segments in which communication is being carried out. In general user-directed checkpointing is a far more effective scheme. When the developer is creating checkpoint locations it is necessary to identify what data should be saved, as data that has not changed from one checkpoint to the next need not be included in the new checkpoint.

The first attempt at the development of fault tolerant MPI applications made use of checkpointing and roll back. Co-Check MPI [9] was the first MPI implementation built that used the Condor [10] library for checkpointing. All process would synchronously checkpoint, this proved to be a drawback with large infrastructures as the procedure could become expensive from a time concern. The result of this work was the creation of a new version of MPI called tuMPI as the modification of the original MPICH implementation was considered too complex. Another similar implementation is Starfish MPI [1] which uses its own system to achieve checkpointing. The use of atomic group communication calls removed the need as in the tuMPI implementation to flush the message queues to avoid messages being lost.

1.4 The Communications World

The essential object in any MPI system is the Communicator or World object through which all communication is carried out. Therefore programs that use only one communicator are more fragile as a failure on one node will rule out the possibility of using global communication routines. In contrast Client/Server architectures are far less prone to a failure having an impact on the system. The failure of a client should have little or no impact on the server. In this architecture communication is carried out in a point to point manner. In the event of a communications failure with a client device, the server can simply cease communication with the client and close all relevant streams.

SETI@home [8] is a perfect example of this where the client devices are used purely for computation, whilst communication is only carried out to return results and retrieve another work unit. Should the server itself go down then the clients can simply try again at a future point in time. The facilitation of several backup servers can ensure the maximum uptime of the server system in case faults occur, to the primary. System state in such systems can be easily maintained. MPI programs may also be structured in a similar manner, this is of course dependent on the application domain of the problem at hand, and is well suited to embarrassingly parallel applications. To achieve this in MPI systems the use of intercommunicators is necessary, thereby establishing two groups of processes with all communication occurring between processes in one group and processes in the other.

The Manager Worker architecture may be applied to several application domains from climate prediction [3] and protein folding [6] to applications such as SETI [8]. The key with the architecture is that the Manager maintains a work pool of tasks to be computed, which are in turn distributed to the worker nodes as they become free. This is a highly suitable architecture for fault tolerant systems as the worker nodes maintain a small amount of state information at any instant. A copy of the work assignment may also remain on the Manager node in case of failure and can therefore be farmed out to another node. Data dependance between workers is not required and no global communication is unnecessary.

One must always take into consideration that there is a trade off between fault tolerance and cost. The more fault-tolerant a system, the higher the costs in terms of saving saving state to persistent storage, error checking and reconstituting the world in the case of system failures. The combination of all these factors can severely impact the overall performance of the system, but sometimes this is necessary when system integrity and data consistency are of utmost importance.

Under standard MPI implementations one typically has high-end reliable hardware, and example is system RAM with in-built checking for consistency. Even on lower-end systems, the networking issues are still not so much of a problem as nodes are generally linked over a high speed wired network. In the wireless domain, network connectivity can be intermittent with nodes coming and going from the network due to physical proximity, and transmission interference.

1.4.1 Communicator Reconfiguration

Fagg and Dongerra [5] discuss four possibilities for the recovery of a communicator that has an error state. To reconstitute a valid communicator it is necessary to rebuild it using modified versions of one of the MPI communicator build functions such as MPI_Comm_create, MPI_Comm_split or MPI_Comm_dup. SHRINK allows the communicator to be reduced in size so that the structure is contiguous, this requires the modification of the ranks. BLANK is similar to shrink, except the communicator now contains blanks instead of references to nodes that are now unavailable. Communications with a gap will result in errors, therefore prohibiting the effective use of global communication functions. REBUILD is the most complex mode available that forces the creation of new processes to fill any gaps until the communicator is fully reconstituted. This mechanism can either fill gaps in the communicator or shrink the communicator and add additional nodes so it returns to its original size. The final option is ABORT which will result in the graceful abortion of the application on detection of an error.

1.5 The Costs of Fault Tolerance

Gropp and Lusk [7] investigated fault tolerance in MPI by dealing with only one probability for a failure of the system. It is assumed that at most one failure may occur between checkpoints with the probability α. Therefore the total run time may be defined by

$$E_T = \frac{T}{t_0} \cdot \left(k_0 + t_0 + \alpha \cdot \left(k_1 \cdot t_0 + \frac{1}{2} t_0^2 \right) \right)$$

where k_0 is the time to create a checkpoint and k_1 is the time to read / restore a checkpoint. Accordingly, the optimal time between checkpoints is given by $t_0 = \sqrt{\frac{2k_0}{\alpha}}$ therefore the expected computation time is $T(1 + \alpha \cdot k_1 + \sqrt{2\alpha \cdot k_0})$.

2 Quantizing the Costs for Mobile Fault Tolerance

Checkpoints may be created at regular intervals to save program state. If T is the total execution time and t_0 is the time between checkpoints then the number of checkpoints is given by $\frac{T}{t_0}$. Under standard MPI characteristics a node may fail due to errors on the device itself, or due to errors related to inter-device communications. In the mobile world these failures have been classified into three distinct categories.

1. Normal failure with the probability α_0.
2. A device fails because it exceeded the Bluetooth range, probability of α_1.
3. A device terminates the application with the probability α_2.

The accurate detection of errors is only one half of the fault tolerance equation, the other is to ensure that applications can carry on from a previously valid system state. This may be achieved through the use of checkpointing. In the case of errors caused by devices moving in and out of the Bluetooth range (10 meters, for a typical phone), one may attempt to restore the original connections (DataInput / DataOutput Streams) as the physical address of the devices in question remains the same. In other cases it may be necessary to re-initialise the world. This would require the carrying out of device and service discovery once again, and the reformation of the network based on the presently active nodes detected by the discovery process. This can be an expensive operation as the combined discovery process can last on the order of eighteen to twenty seconds. The following classifications may be used to determine the time constraints of checkpointing and restoration.

k_0 = the time to create a checkpoint
k_1 = the time to read / restore a checkpoint
k_2 = the time to restore the communication channels
k_3 = the time to initialise the world

2.1 Single-Point Check and Recovery

There are three distinct cases to be evaluated as mentioned previously with regard to possible failures. The overall cost of fault tolerance is a combination of these three together. The process of checkpointing will occur at regular time intervals of t_0.

2.1.1 Type 1 Failures (NORMAL_FAILURE)

This is when a normal failure occurs. A normal failure may be considered to be a standard communication error. The costs involved are to create a checkpoint, read the previous checkpoint and restore the state hence the equation is exactly as in Gropp & Lusk [7]. For one checkpoint we have

$$E_1 = (1 - \alpha_0 \cdot t_0)(k_0 + t_0) + \alpha_0 \cdot t_0 \cdot \left(k_0 + t_0 + k_1 + \frac{1}{2}t_0\right) =$$
$$= k_0 + t_0 + \alpha_0 \cdot \left(k_1 \cdot t_0 + \frac{1}{2}t_0^2\right) = k_0 + t_0 \cdot (1 + \alpha_0 \cdot k_1) + \frac{1}{2}\alpha_0 \cdot t_0^2$$

which gives the following cost over $\frac{T}{t_0}$ checkpoints

$$E_1^t(t_0) = \frac{T}{t_0}\left[k_0 + t_0(1+\alpha_0 \cdot k_1) + \frac{1}{2}\alpha_0 \cdot T_0^2\right]. \tag{1}$$

2.1.2 *Type 2 Failures (RANGE_FAILURE)*

When a device is outside of the Bluetooth network range. The device should return to within the network range and consequently re-establish the I/O connections with the MMPI world. The device will then read the previous checkpoint and restore the system state. Therefore, the total cost for one checkpoint is

$$E_2 = (1-\alpha_1 \cdot t_0)(k_0+t_0) + \alpha_1 \cdot t_0 \cdot \left(k_0 + t_0 + k_2 + k_1 + \frac{1}{2}t_0\right) =$$
$$= k_0 + t_0 + \alpha_1 \cdot t_0 \cdot (k_2+k_1) + \frac{1}{2}\alpha_1 \cdot t_0^2 = k_0 + t_0 \cdot [1 + \alpha_1 \cdot (k_2+k_1)] + \frac{1}{2}\alpha_1 \cdot t_0^2$$

with the total cost given by

$$E_2^t(t_0) = \frac{T}{t_0} \cdot \left[k_0 + t_0\left[1+\alpha_1 \cdot (k_2+k_1)\right] + \frac{1}{2}\alpha_1 \cdot t_0^2\right] \tag{2}$$

2.1.3 *Type 3 Failures (NODETERM_FAILURE)*

When the device itself terminates the application for some reason. In this case all the devices should start the application from scratch which gives the following total cost.

$$E_3 = (1-\alpha_2 \cdot t_0)(k_0+t_0) + \alpha_2 \cdot t_0 \cdot (k_0+t_0+T) = k_0 + t_0 + \alpha_2 \cdot t_0 \cdot T. \Rightarrow$$

$$E_3^t(t_0) = \frac{T}{t_0} \cdot \left[k_0 + t_0(1+\alpha_2 \cdot T)\right]. \tag{3}$$

Total cost over the three types of failure case is

$$E^t(t_0) = E_1^t(t_0) + E_2^t(t_0) + E_3^t(t_0) =$$
$$= \frac{T}{t_0} \cdot \left[3k_0 + t_0 \cdot [3 + \alpha_0 \cdot k_1 + \alpha_1 \cdot (k_2+k_1) + \alpha_3 \cdot T] + \frac{1}{2}(\alpha_0+\alpha_1)t_0^2\right] =$$
$$= \frac{3T \cdot k_0}{t_0} + \frac{T}{3}(\alpha_0+\alpha_1)t_0 + [3 + \alpha_0 \cdot k_1 + \alpha_1 \cdot (k_2+k_1) + \alpha_2 T] \cdot T.$$

The optimal time t_0 between checkpoints can be calculated as follows

$$\frac{dEt}{dt_0} = -\frac{3T \cdot k_0}{t_0^2} + \frac{T}{2} \cdot (\alpha_0+\alpha_1) = 0 \Rightarrow \frac{3T \cdot k_0}{t_0^2} = \frac{T}{2} \cdot (\alpha_0+\alpha_1)$$
$$\Rightarrow t_0^2 = \frac{6k_0}{\alpha_1+\alpha_2} \Rightarrow t_0 = \sqrt{\frac{6k_0}{\alpha_1+\alpha_2}}$$

which gives the following optimal run time

$$\begin{aligned} E^t(t_0) &= \frac{3T \cdot k_0}{\sqrt{\frac{6k_0}{\alpha_1+\alpha_2}}} + \frac{T}{2} \cdot (\alpha_1 + \alpha_2) \cdot \sqrt{\frac{6k_0}{\alpha_1 + \alpha_2}} \\ &\quad + T \cdot [3 + \alpha_0 \cdot k_1 + \alpha_1 \cdot (k_2 + k_1) + \alpha_2 \cdot T] \\ &= \sqrt{\frac{3}{2}} \cdot \sqrt{k_0 \cdot (\alpha_1 + \alpha_2)} \cdot T + \sqrt{\frac{3}{2}} \cdot \sqrt{k_0 \cdot (\alpha_1 + \alpha_2)} \cdot T \\ &\quad + T \cdot [3 + \alpha_0 \cdot k_1 + \alpha_1 \cdot (k_2 + k_1) + \alpha_2 \cdot T] \\ &= \left[\sqrt{6k_0(\alpha_1 + \alpha_2)} + 3 + \alpha_0 \cdot k_1 + \alpha_1 \cdot (k_1 + k_2) + \alpha_2 \cdot T\right] \cdot T. \end{aligned}$$

$E^t(t_0)$ represents the minimal time that may be achieved with mobile fault tolerance. Consequently as α_0, $\alpha_2 \simeq 0$ the highest probability for error is that of a device moving outside of the range α_1 therefore the optimal time in this case may be given by $E^t(t_0) = \left[\sqrt{6k_0 \cdot \alpha_1} + 3 + \alpha_1 \cdot (k_1 + k_2)\right] \cdot T$.

2.2 Multi-Point Check and Recovery

Due to the probability that different failures can occur with varying degrees of certainty the process of checkpointing and recovery may be carried out at different time intervals according to the type of failure and the likelihood of the failure occurring. The three cases in question are Normal failures such as general I/O errors, Range failures where by a node may move outside the range of another device and Node Termination failures that result in the application or the device itself being shut down.

Type 1 Failures (NORMAL_FAILURE)	$\Rightarrow \alpha_0$ probability of a failure
	$\Rightarrow t_0$ time to test
Type 2 Failures (RANGE_FAILURE)	$\Rightarrow \alpha_1$ probability of a failure
	$\Rightarrow t_1$ time to test
Type 3 Failures (NODETERM_FAILURE)	$\Rightarrow \alpha_2$ probability of a failure
	$\Rightarrow t_2$ time to test

2.2.1 Type 1 Failures (NORMAL_FAILURE)

A failure such as this could easily happen in the wireless environment, where the transmission of a message could be disrupted by interference from the environment, resulting in the loss of the message data. The estimated cost for this operation can be defined by

$$\begin{aligned} E_1 = (1 - \alpha_0 \cdot t_0)(k_0 + t_0) + \alpha_0 \cdot t_0 \cdot \left(k_0 + t_0 + k_1 + \frac{1}{2}t_0\right) = \\ = k_0 + t_0 \cdot (1 + \alpha_0 \cdot k_1) + \frac{1}{2}\alpha_0 \cdot t_0^2 \end{aligned}$$

which gives the following cost over $\frac{T}{t_0}$ checkpoints

$$E_1^t(t_0) = \frac{T}{t_0} \cdot \left[k_0 + t_0 \cdot (1 + \alpha_0 \cdot k_1) + \frac{1}{2}\alpha_0 \cdot T_0^2\right]. \tag{4}$$

2.2.2 Type 2 Failures (RANGE_FAILURE)

When a few devices are used in conjunction with one another the probability of a range failure is usually very small as the people who initiated the application typically remain within close proximity to one another. Examples of this interaction could be a few people playing a multiplayer game while waiting for a bus, another example could be the exchange of contact details between business men at a board meeting. In general connections between Bluetooth computing devices last for a limited time period. In the case of peripheral devices such as keyboards and mice, they almost always remain within a few feet of the controlling system.

The time in this case is a combination of the time to create a checkpoint, the time to test for a range failure, the restoration time of the lost communication channel as well as the time to read / restore a checkpoint, which is determined by the following equation

$$\begin{aligned} E_2 = (1-\alpha_1 \cdot t_1)(k_0+t_1) + \alpha_1 \cdot t_1 \cdot \left(k_0 + t_1 + k_2 + k_1 + \frac{1}{2}t_0\right) = \\ = k_0 + t_1 \cdot [1 + \alpha_1 \cdot (k_2 + k_1)] + \frac{1}{2}\alpha_1 \cdot t_0^2 \end{aligned}$$

with the total cost given by

$$E_2^t(t_1) = \frac{T}{t_1} \cdot \left[k_0 + t_1 \left[1 + \alpha_1(k_2+k_1)\right] + \frac{1}{2}\alpha_1 \cdot t_0^2\right] \tag{5}$$

2.2.3 Type 3 Failures (NODETERM_FAILURE)

Type 3 failures indicate the complete cessation of an application / device. Some of the causes of this could be user intervention, the termination of Bluetooth communications on the device to conserve power, or even a complete power loss.

$$E_3 = (1-\alpha_2 \cdot t_2)(k_0+t_2) + \alpha_2 \cdot t_2(k_0 + t_2 + T) = k_0 + t_2 + \alpha_2 \cdot t_2 \cdot T. \Rightarrow$$

$$E_3^t(t_2) = \frac{T}{t_2} \cdot \left[k_0 + t_2 \cdot (1 + \alpha_2 \cdot T)\right]. \tag{6}$$

The overall cost is a combined estimated running time for the previous three tests to run in conjunction within the application

$$E^t(t_0, t_1, t_2) = E_1^t(t_0) + E_2^t(t_1) + E_3^t(t_2).$$

The minimum overall costs involved for the testing of the three distinct possibilities of failure may be defined as the minimum time for each individual operation combined. It is clear that

$$\min_{t_0,t_1,t_2} = E^t(t_0,t_1,t_2) = \min_{t_0} E_1^t(t_0) + \min_{t_1} E_2^t(t_1) + \min_{t_2} E_3^t(t_2).$$

In the following the three minimum values are evaluated.

Lemma 1

The minimum of $E_1^t(t_0)$ is

$$\min_{t_0} E_1^t(t_0) = T \cdot \left(1 + \alpha_0 \cdot k1 + \sqrt{2 \cdot \alpha_0 \cdot k_0}\right)$$

and is achieved by

$$t_0 = \sqrt{\frac{2k_0}{\alpha_0}}$$

This is very like the optimal time between checkpoints calculated by Gropp & Lusk [7] in which they differentiated with respect to t_0, yielding a result of $t_0 = \sqrt{\frac{2k_0}{\alpha}}$.

Lemma 2

$$\min_{t_1} E_2^t(t_1) = T \cdot \left[1 + \alpha_1 \cdot (k_1 + k_2) + \sqrt{2 \cdot \alpha_1 \cdot k_0}\right]$$
$$t_1 = \sqrt{\frac{2k_0}{\alpha_1}}$$

Proof 3

We can differentiate to find the minimum t_1.

$$\frac{dE_2^t(t_1)}{dt_1} = \left[\frac{T \cdot k_0}{t_1} + T \cdot (1 + \alpha_1 \cdot (k_1 + k_2)) + \frac{T \cdot \alpha_1 \cdot t_1}{2}\right] = -\frac{T \cdot k_0}{t_1^2} + \frac{T \cdot \alpha_1}{2} = 0 \Rightarrow$$
$$\Rightarrow \frac{k_0}{t_1^2} = \frac{\alpha_1}{2} \Rightarrow t_1^2 = \frac{2k_0}{\alpha_1} \Rightarrow t_1 = \sqrt{\frac{2k_0}{\alpha_1}}$$

The minimum value of $E_2^t(t_1)$ is given by

$$\min E_2^t(t_1) = E_2^t\left(\sqrt{\frac{2k_0}{\alpha_1}}\right) == \frac{T \cdot k_0}{\sqrt{\frac{2k_0}{\alpha_1}}} + T \cdot (1 + \alpha_1 \cdot (k_1 + k_2) + \frac{T \cdot \alpha_1}{2} \cdot \sqrt{\frac{2k_0}{\alpha_1}} =$$
$$= T \cdot \frac{k_0 \cdot \alpha_1}{2} + T \cdot [1 + \alpha_1 \cdot (k_1 + k_2)] + T \cdot \sqrt{\frac{\alpha_1 \cdot k_0}{2}} = T \cdot \left[\sqrt{2k_0 \cdot \alpha_1} + 1 + \alpha_1 \cdot (k_1 + k_2)\right] =$$
$$= T \cdot \left[1 + \alpha_1 \cdot (k_1 + k_2) + \sqrt{2\alpha_1 \cdot k_0}\right]$$

Lemma 4

The minimum value of $E_3^t(t_2)$ is

$$\min_{t_2} E_3^t(t_2) = k_0 + T \cdot (1 + \alpha_2 \cdot T)$$

and is given by $t_2 = T$.

Proof 5

Differentiating $E_3^t(t_2)$ we find

$$\frac{dE_3^t(t_2)}{dt_2} = \left[\frac{T \cdot k_0}{t_2} + T \cdot (1 + \alpha_2 \cdot T\right]' = -\frac{T \cdot k_0}{t_2^2} < 0$$

so $E_3^t(t_2)$ is decreasing. This means that the minimum value is achieved for $t_2 = T$, hence

$$\min_{t_2} E_3^t(t_2) = E_3(T) = k_0 + T \cdot (1 + \alpha_2 \cdot T).$$

In conclusion the overall minimal cost becomes

$$\min_{t_0,t_1,t_2} E^t(t_0,t_1,t_2) = T \cdot \left[1 + \alpha_0 \cdot k_1 + \sqrt{2\alpha \cdot k_0}\right]$$
$$+T \cdot \left[1 + \alpha_1 \cdot (k_1 \cdot k_2) + \sqrt{2\alpha_1 \cdot k_0}\right] + k_0 + T \cdot (1 + \alpha_2 \cdot T).$$

The overall minimum times between the Single-Point and Multi-Point Check and Recovery systems are compared to determine the most efficient approach. Thus providing a basis for the selection of the best approach for a particular application requiring fault tolerance.

$$E_1 = T\left[\sqrt{6k_0 \cdot (\alpha_0 + \alpha_1)} + 3 + \alpha_0 \cdot k_1 + \alpha_1(k_1 + k_2) + \alpha_2 \cdot T\right]$$
$$E_2 = k_0 + T \cdot \left[3 + \alpha_0 \cdot k_1 + \alpha_1 \cdot (k_1 + k_2) + \alpha_2 \cdot T + \sqrt{2k_0} \cdot [\sqrt{\alpha_0} + \sqrt{\alpha_1}]\right]$$
$$E_2 - E_1 = k_0 + T \cdot \left[\sqrt{2k_0}\,[\sqrt{\alpha_0} + \sqrt{\alpha_1}] - \sqrt{6k_0 \cdot (\alpha_0 + \alpha_1)}\right]$$
$$= k_0 + T \cdot \sqrt{2k_0} \cdot \left[\sqrt{\alpha_0} + \sqrt{\alpha_1} - \sqrt{3 \cdot (\alpha_0 + \alpha_1)}\right].$$

If $E_2 - E_1 > 0$ is evaluated we find.

$$k_0 + T \cdot \sqrt{2k_0} \cdot \left[\sqrt{\alpha_0} + \sqrt{\alpha_1} - \sqrt{3 \cdot (\alpha_0 + \alpha_1)}\right] > 0.$$
$$k_0 + T \cdot \sqrt{2} \cdot \left[\sqrt{\alpha_0} + \sqrt{\alpha_1} - \sqrt{3 \cdot (\alpha_0 + \alpha_1)}\right] > 0.$$
$$\sqrt{k_0} > T \cdot \sqrt{2} \cdot \left[\sqrt{3 \cdot (\alpha_0 + \alpha_1)} - \sqrt{\alpha_0} - \sqrt{\alpha_1}\right]$$

Theorem 6
If the overall execution time T satisfies

$$T < \frac{\sqrt{k_0}}{\sqrt{2}\left[\sqrt{3 \cdot (\alpha_1 + \alpha_2)} - \sqrt{\alpha_0} - \sqrt{\alpha_1}\right]} \tag{7}$$

then $E_2 > E_1$ hence the Single-Point Check and Recovery strategy is better. This theorem says that if the execution time is smaller than a threshold then the first approach is more effective.

Theorem 7
If the overall execution time is greater

$$T > \frac{\sqrt{k_0}}{\sqrt{2}\left[\sqrt{3 \cdot (\alpha_1 + \alpha_2)} - \sqrt{\alpha_0} - \sqrt{\alpha_1}\right]} \tag{8}$$

then $E_2 < E_1$ hence the Multi-Point Check and Recovery strategy should be chosen over the latter approach.

3 Conclusion

A principle requirement for the creation of a fault-tolerant system is the ability to detect errors. Fault tolerance in relation to ad-hoc wirelessly interconnected mobile devices is a far more complex task than fault tolerance for high end clusters and parallel machines with fixed cabled infrastructure. Within the Bluetooth world the detection of a failure does not immediately mean that a node has completely failed / terminated. It is highly possible for a node to simply move out of the range of the Bluetooth Radio of another device, and as it moves out it can just as easily move back into range. Therefore it is necessary to allow for this eventuality, and attempt to re-establish the connection. If however, after a number of attempts to re-establish the connection, one must consider that the node in question has had a critical system failure, rendering the node completely cut off from the interconnect of the mobile parallel world. Hence this eventuality must be reported to the parallel world, so it can be reconfigured as appropriate, be it in the form of a recovery mechanism or termination of the parallel application.

Given the two checkpointing strategies previously described a developer may opt for one strategy over another based on the expected overall execution time of the application in question. If $E_2 > E_1$ then the Single-Point Check and Recovery strategy is best, otherwise the Multi-Point approach is more suitable.

Acknowledgements

This research project was funded under the "Irish Research Council for Science, Engineering and Technology" funded by the "National Development Plan".

References

1. A. Agbaria and R. Friedman (1999), *Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations*, The Eighth International Symposium on High Performance Distributed Computing, pp. 167–176.
2. Bluetooth-SIG (2001), *Annex A (Normative): Timers and Constants* Bluetooth Specification Version 1.1.
3. *Climate Prediction.net* (2008), `http://climateprediction.net`
4. D. C. Doolan, S. Tabirca, and L. T. Yang (2006), *Mobile Parallel Computing*, 5th International Symposium on Parallel and Distributed Computing (ISPDC06), pp. 161–167.
5. G. E. Fagg and J. J. Dongarra (2000), *FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World*, Lecture Notes in Computer Science, Vol. 1908, pp. 346-353.
6. Folding@home (2008), *Distributed Computing, understand protein folding, misfolding, and related diseases*, `http://folding.stanford.edu`
7. W. Gropp and E. Lusk (2002), *Fault Tolerance in MPI Programs*, Cluster Computing and Grid Systems Conference, `http://www-unix.mcs.anl.gov/~gropp/bib/papers/2002/mpi-fault.pdf`
8. SETI@Home (2008) *The Search for Extra Terrestial Intelligence at Home*, `http://setiathome.ssl.berkeley.edu`
9. G. Stellner (1996), *CoCheck: Checkpointing and Process Migration for MPI*, Parallel Processing Symposium, pp. 526–531.
10. T. Tannenbaum and M. Litskow (1995), *Checkpoint and Migration of Unix Processes in the Condor Distributed Processing System*, Dr. Dobbs Journal, vol. 227, pp. 40–48.