Abstract

The DIR net (detection-isolation-recovery net) is the main module of a software framework for the development of embedded supercomputing applications. This framework provides a set of functional elements, collected in a library, to improve the dependability attributes of the applications (especially the availability). The DIR net enables these functional elements to cooperate and enhances their efficiency by controlling and co-ordinating them. As a supervisor and the main executor of the fault tolerance strategy, it is the backbone of the framework, of which the application developer is the architect. Moreover, it provides an interface to which all detection and recovery tools should conform. Although the DIR net is meant to be used together within this fault tolerance framework, the adopted concepts and design decisions have a more general value, and can be applied in a wide range of parallel systems.

1. Introduction

Fault tolerance, or the ability to tolerate faults, is a crucial requirement for any successful industrial product. It is intended to increase system properties as availability, maintainability etc., commonly referred to as dependability [1]. Many embedded applications require high availability and high performance computing which may only be provided by parallel computing systems. However, an inherent weak point of parallel computing is the higher probability of failures. This is not only due to the multiplicity of hardware, but also and mainly to the higher complexity of the software; this problem is enlarged by the industrial environment and the long execution time. But at the same time, parallelism can be exploited to provide fault tolerance, making use of the available redundancy.

Fault tolerance can be achieved by using specialised hardware, fault-tolerant operating systems and/or by the development of fault-tolerant software. The hardware approach is the most expensive one, though the software approach may also be expensive if the application developers have to design and integrate fault tolerance strategies from scratch. Moreover, ad hoc solutions can be an impairment for portability and maintenance.

Within the ESPRIT project EFTOS (Embedded Fault-Tolerant Supercomputing) [2, 3], several embedded applications are targeted which require both high performance computing and fault tolerance. These different applications have similar requirements with respect to fault tolerance and can benefit from identical solutions. The project's goal is to streamline these solutions into a reusable fault tolerance framework that can be flexibly and easily integrated into the target applications. The framework consists of a set of tools collected in a fault tolerance library. The developers can select and customise those tools that are necessary for the target application.

The EFTOS framework does not rely on specific hardware features to achieve fault tolerance, unless those features are found in most common parallel computers. This is in contrast to commercial products like SUN IMP [4], SDFT and a scientific project like CHIMERA [5]. As it focuses on embedded parallel computing, it does not benefit from time-consuming techniques like checkpointing and rollback, file replication and backward error recovery in general as is used for example in ISIS [6], HATS [7] and Fail-safe PVM [8]. The framework focuses on soft real-time, embedded applications.
such a framework, development costs and time-to-market are significantly lowered.

The rest of the paper is structured as follows. Section 2 describes the target platform on which the framework was developed and tested. A concise overview of the library is presented in Section 3. Section 4 is devoted to the architecture of the DIR net and its fault tolerance strategy. The paper is concluded by an outlook to future developments.

2. Description of the target platform

The target platform is a Parsytec CC system [9] which combines powerful processing nodes, I/O modules and routers into a parallel system with an MIMD architecture. Every node is based on a standard PowerPC 604 processor with 32-128 MB local memory and may have its own disc as well as other I/O units. HS-links (HS = high speed) with a link speed of up to 1 Gbit/s are the primary means of communication between nodes. They connect the nodes via a PCI interface and thin coaxial cables to the hardware routers (8-way routing). A control net is available for system monitoring and control.

The parallel operating system, EPX (Embedded Parallel extensions to UniX), provides the functionality to operate the parallel environment, using a message-passing API. For the operation with node local resources, it uses services of the underlying node local operating system. EPX offers both static and dynamic facilities. The initialisation phase of the programming model embedded in EPX includes an initial booting and loading mechanism which distributes an identical main program to all processors. A set of global data kept on each node is used for the identification of the ‘own’ position within the system, which allows the execution of different sections of the program and the use of specific topologies. Communication is modelled on the base of virtual links, which build point-to-point connections between arbitrary threads within the topology. At run-time, light-weight processes -threads- can be created within an EPX program. A bundle of created threads, residing on the same node, are running concurrently in the same context and share global variables. It is also possible to load and run additional code at any time, thereby creating a new context.

For efficient fault tolerance, some support of the kernel is required. The kernel was therefore extended with application restart at single nodes, non-blocking receive, stopping/restarting of virtual links and the possibility of preventing a thread from being rescheduled. These features are also available in many other parallel environments. Some functionality like the remote creation of threads and signal communication between threads was missing in EPX. These features have been implemented in the EFTOS FT-library on a high level, via a Server net (see Section 3).

Portability has been successfully tested via the actual porting to other operating systems and other hardware (e.g. Windows NT on Intel Pentium platform, TEX on DEC Alpha platform). For missing or different features, an adaptation layer was designed.

3. Description of the library

The library is focused on tolerating data-dependent software errors. The strategy is the following [10]:
1. error detection — recognise that an error has occurred;
2. fault diagnosis — determine which specific module produced the error;
3. error isolation — prevent error propagation and isolate unusable parts;
4. error recovery — regain operational status or restore the system's integrity.

The library provides tools for each of these steps. It is a modular FT-library, i.e., the user can select the necessary amount (and the corresponding overhead) of fault tolerance, by using the appropriate tools. The library is layered between the application and the underlying OS and hardware.

According to its architecture, the FT-library consists of 4 types of elements (Figure 1):

- The Server net provides necessary extensions to the operating system's functionality, namely: remote creation of threads, sending signals to threads, gathering information on the application's behaviour (creation and removal of threads, links, etc.). The Server network has a distributed architecture.
consisting of a server thread on each node, and is fully interconnected. Each thread on a node is connected to the local server thread, and can send and receive commands and messages via this interconnection. The main role of the Server network is hence to forward information through the system.

- The DIR net is responsible for the co-ordination of error detection, diagnosis, isolation and recovery. It has a hierarchical structure, consisting of one manager residing on a particular node in the allocated partition and an agent on every other node. Some specialised agents, called "backup agents", can take over the role of manager.

- The detection tools, abbreviated as Dtools, detect errors and report the type and the location of the error to the DIR net. Each type of Dtool is capable of detecting a specific kind of error. Examples are a watchdog timer to detect time-outs, a trap handler to catch signals, etc.

- FT-mechanisms are more complex: some applications require specific recovery actions, e.g. via atomic actions. The FT-mechanisms therefore do more than error detection, but still report to the DIR net to make sure that the DIR net's view on the system is complete. Examples include an atomic action tool, a voting tool and stable storage.

This results in the following strategy: the DIR net makes use of Dtools and FT-mechanisms to detect errors. The location and the type of the error is known because this is diagnosed by the Dtool or FT-mechanism. Error isolation is achieved by disabling all the links between the thread (node) where the error occurred and all the connected threads (nodes). At that point, the fault is assumed to be contained. The error recovery is determined in the following order:

1. If the Dtool that detected the error proposes a recovery action, this action will be executed.
2. If the particular error is mentioned in the user's recovery file (see Section 4.2), the corresponding recovery rules are executed. To this end, an interpreter has been developed [11, 12].
3. Otherwise, a default recovery action (e.g. node restart) is executed.

A recovery rule, which is supplied by the application developer, can consist of one or more of the following actions:

- warning of a thread, a node, a group of threads;
- killing of a thread, a node, a group of threads;
- restarting a thread, a node, a group of threads;

- starting of a new thread (which can also be a user-supplied recovery tool).

FT-mechanisms can provide additional specialised recovery actions.

4. Architecture of the DIR net

This Section describes the most complex module of the library, the DIR net, in more detail. The requirements for this module are:

- it must be self-fault-tolerant; for instance, it must be able to tolerate a crash of the manager or of the node on which it resides (in this case a backup agent takes over its role);
- it should be transparent to the user;
- it must have a global view of the system, down to the placement of every thread, its interconnections, its start-up function and arguments, its status, etc.;
- it must be fast; this is achieved by keeping all data in memory for fast access, by reducing inter-node communication, by avoiding memory re-allocation at run-time, ...

4.1 Description of the architecture

In Figure 2, a graphical representation of the DIR net architecture is shown. As can be seen, the DIR net
Figure 3: *I'm alive*-mechanism, part 1.

consists of 6 types of modules, each consisting of exactly one thread:

- A **normal agent** is connected to the manager and every backup agent. It has only a view on node-local threads and on threads that are directly connected to it.

- The **manager** is connected to every agent and uses asynchronous, non-blocking, buffered communication. It has a system wide overview and takes all the important decisions. It is the single executor of system-wide actions as rebooting and restarting.

- A **backup agent** behaves as a normal agent, but can replace a manager when the latter fails. It receives the same input as the manager and has the same overview, but it does not take decisions.

- The **input and output buffers** provide a backup agent or manager with fault-tolerant communication functions. It allows asynchronous communication, increasing performance.

- An **<I'm alive>-thread** is used to detect node crashes. It only comes into action when the local agent or manager has crashed, in which case it starts producing *<I'm alive>*-messages to inform the rest of the DIR net that a module failed on that node, while the node itself is still alive.

- During recovery, the DIR net consists of an additional module, the **recovery thread**, which invokes all necessary recovery actions, according to a global, a local or a user-defined strategy. It is created by the manager after the first error is detected and ends when the system has recovered from all errors.

4.1.1 The database

Every action in the system,—creation and removal of threads and links, and a change of their status—is monitored by the Server net and reported to the DIR net. This monitoring is achieved by substituting the standard EPX commands with similar framework functions that send a message to the local server when they are used. The manager stores this information in its database, to be used when diagnosis, isolation and recovery actions are necessary. This information includes, e.g., arguments for thread creation, detected errors, ongoing recovery measures. ... The backup agents make sure that they maintain a consistent copy of this database. A separate class of functions has been developed to set-up, update and query the database and to compare databases on different nodes via checksums and update counts, and to check consistency. The normal agents also keep a database, but only concerning node-local threads and links. The complete database is kept in local memory to increase speed.

4.1.2 The **<I'm alive>-mechanism**

To detect complete node crashes, an *<I'm alive>*-mechanism has been implemented, consisting of two parts, one part to detect agent crashes, the other part to detect a manager crash.

The first part is depicted in Figure 3. The manager expects a message from every agent within a deadline. This message can be a normal message or, if no messages are available to be sent to the manager in piggybacking, a dummy message. If a time-out expires for a certain agent, the manager has to discover whether the entire node crashed or only that agent. That is why an *<I'm alive>*-thread is used. When the deadline for sending a normal or dummy message passes for a particular agent, the *<I'm alive>*-thread notices this and autonomously
Figure 5: The <I'm alive>-mechanism, part 2.

 sends an <I'm alive>-message to the manager. The manager then knows that the agent has crashed, but not the complete node. If there is no message at all after a second time-out, it is assumed that the complete node has crashed. The timing scheme is shown in Figure 4.

The second part, depicted in Figure 5, uses a similar approach to detect a manager crash. The input buffers of the backup agents expect a message from the manager when these input buffers have been empty for a certain period. This message contains the checksum and an update count of the manager's database. When the manager has crashed, they expect an <I'm alive>-message from the <I'm alive>-thread. The absence of any message is interpreted as a node crash. When there is a difference between the received checksum or update count and the local ones, a protocol is started to decide who is faulty and who needs to be restarted.

4.1.3 The state file

When an agent or an entire node is restarted, the new agent retrieves a copy of the database from the manager, and the agent brings itself in a state that is consistent with the rest of the DIR net and the Dtools, e.g. by restoring the connections. An agent's initialisation procedure is hence different in these three cases: if the application is started, if the agent is restarted or if the entire node is restarted. To this end, a state file is kept on disk where the configuration (manager, backup agent or normal agent) and its runlevel (OK, starting, restarting, ...) is stored for each node. Special functions have been developed for updating and consulting this state file, using read- and write-locks to avoid conflicts, integrating a time-stamp.

4.1.4 The Monitor

The transparency of the DIR net is an advantage for the user, but for the operator it is important to know what happens on the system. Therefore, a hypermedia monitor [13] has been developed that represents the state of the entire system, e.g. type, location and status of all threads and links, detected errors, recovery measures, configuration of the fault tolerance framework, etc. The monitor is connected to the manager, and is informed on every important action which it shows on screen. The user can choose the amount of detail that needs to be displayed to allow fast interpretation. This monitor can also be used to debug the user application, to inject faults into it and to assess the effectiveness of the selected set-up for the fault tolerance framework.

4.2 Fault tolerance strategy of the DIR net

When the application developer starts the DIR net, the amount and the location of backup agents is chosen. The backup agents will autonomously elect the manager. The developer has the freedom to select which Dtools and FT-mechanisms are used, when they are started, where they are located, how they are configured and to which agents they are connected. He/She can also write his/her own Dtools and hook them to the DIR net; for this reason a fixed protocol and API has been designed to which all Dtools adhere. This configuration can be changed at run-time. The same flexibility applies for the recovery actions: the application developer can supply a recovery file that contains the recovery action(s) that should be applied when a particular error is detected at a particular location and moment in the system. This file has to be coded according to a simple and yet effective grammar [11].

4.2.1 Detection

When an error is detected, the Dtool sends a message to a connected agent. This contains the type of the error (e.g. division by zero, time-out, ...), its location (e.g. which thread, group, node or link) and a proposal for a recovery action. The agent forwards this message through the DIR net to the manager and all the backup agents which update their database.

4.2.2 Diagnosis & isolation

Diagnosis is started by the Dtool or FT-mechanism as they point out the location where the error was observed. Then, the manager isolates the concerned thread, group or node. One error though can trigger new errors, and the manager continues isolating threads and nodes until no new errors occur. Isolation of a thread or a node is achieved by sending a list to the local agent containing the links that have to be stopped. When a thread or node has no more active links to communicate with, it is isolated.

After the isolation, each backup agent and the manager have a list of errors that were detected. Each entry contains the type of the error, location and proposed recovery action (if any). The lists might not have the same order on each node, and there might be duplicate entries, but this is dealt with during recovery.

4.2.3 Recovery

As soon as the first error is detected, the manager spawns the recovery thread. This thread invokes all necessary recovery actions, while the manager and the
backup agents receive the return values of these recovery actions and adjust their databases accordingly.

For each error, the recovery thread checks if there is a proposed recovery action by the Dtool and executes it. If it was successful, the error is removed from the error lists.

Next, the recovery thread scans the recovery file, and executes corresponding recovery actions.

Finally, if all applicable recovery rules have been executed and there are still errors left, the default action is executed for every error. This default action is also defined in the recovery file as: ‘kill’ or ‘restart’ the faulty entity.

4.3 Experiences

The DIR net has been used as a backbone of the EFTOS framework. This framework further comprises basic tools and mechanisms for fault tolerance (detection, isolation, recovery, fault masking, etc.) and a high-level language (RL) to specify recovery strategies. RL allows the application developer to indicate which fault tolerance actions need to be executed by the backbone through its basic elements. As such, RL provides a second application layer which separates the non-functional description of the application from the functional one [12]. This divide-and-conquer approach increases the maintainability of the fault-tolerant application.

This EFTOS approach has been integrated into two industrial applications: an image processing application from the postal automation area [14], and a sequence controller of the energy transport area [15].

Measurements on different platforms allowed to obtain time and resource overheads. These concern the different elements of the framework in stand-alone configuration, integrated framework measurements and application measurements. For the tested industrial target applications, time overhead is typically 5%, while the library size increases the size of the application of typically 5-10%.

This evaluation further incorporated an analytical deduction of the appropriateness of the framework elements, starting from the overhead measurements in a given implementation [16].

5. Conclusion

The DIR net, the Server net and several Dtools and FT-mechanisms have been tested by the industrial project partners. These industrial application developers validate the framework and provide feedback. According to them, the greatest strength of the fault tolerance framework lies in its flexibility, adaptability, and in its ease of use. The developer can choose the degree of fault tolerance that is needed, based on which faults are likely to occur, and can determine the appropriate recovery actions that should be taken, based on the type and structure of the application. Application-specific detection and recovery tools, which have to conform to the DIR net interface, can be easily integrated. A recovery file allows to define the recovery strategy that are to be executed by the DIR net. Assessment of the chosen strategy can be achieved by making use of the fault injection and monitoring capabilities of the monitor.

The overhead of the DIR net is dependent on the developer's configuration. The application developer can change the number of backup agents, the size of the buffers, the frequency of the <I'm alive>-messages, the number and type of Dtools and fault tolerance mechanisms, and so on. Because the monitor also displays time values, he/she is able to quickly assess the trade-off.

In the near future, the framework will be extended to fulfil real-time requirements.

Acknowledgements. This project is partly sponsored by an FWO Krediet aan Navorsers, by the Esprit-IV Projects 21012 EFTOS, 28620 TIRAN and by COF/96/11. Geert Deconinck is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium). Rudy Lauwereins is a Senior Research Associate of the Fund for Scientific Research - Flanders (Belgium).

6. References


transparent recovery" (Carnegie Mellon University, School of Computer Science, Feb. 1993).


