OFTT: A Fault Tolerance Middleware Toolkit for Process Monitoring and Control Windows NT Applications

Myron Hecht, Xuegao An, Bing Zhang, Yutao He
SoHaR Incorporated
8421 Wilshire Blvd. Suite 201
Beverly Hills, CA, 90211
{myron,xuegao,bzhang,yutao}@sohar.com

Abstract

This paper describes the OFTT (OLE Fault Tolerance Technology), a fault tolerance middleware toolkit running on the Microsoft Windows NT operating system that provides required fault tolerance for networked PCs in the context of industrial process monitoring and control applications. It is based on the Microsoft Component Object Model (COM) and consists of components that performs checkpoint-saving, failure detection, recovery, and other fault tolerance functions. The ease with which this technology can be incorporated into one application represents the primary innovation. It is hoped that by making fault tolerance more compatible with standard software architectures, more reliable PC-based monitoring and control systems can be built conveniently.

1. Introduction

Windows NT-based PCs have become popular platforms in industrial process monitoring and control applications such as SCADA (Supervisory Control And Data Acquisition) [1]. A typical configuration in this context is a PC in the control room connected to a number of Programmable Logic Controllers (PLCs) on the plant floor via an industrial automation network. A PLC interfaces with various types of input/output devices (such as sensors, valves), reads inputs, processes data, and generates corresponding control outputs. In the meantime, data are sent to the PC where they will be further processed, stored, and made available for other applications [2]. Historically, hardware vendors define proprietary data formats in developing device drivers. An application that needs to access these data has to resolve the format inconsistency by itself independently. As a result, development and maintenance of application software is a difficult and time-consuming task. To solve the problem, a standard software architecture, *OLE for Process Control* (OPC) [3] has been developed by the industrial process control community. OPC is based on Microsoft's OLE/COM [4] technology and specifies a unified interface for accessing different types of data. A hardware vendor encapsulates details of the device driver into a COM object (called *OPC server*) that provides standard interfaces (called *OPC interfaces*) to any application (called an *OPC client*) in a consistent manner.

The OPC standard does not address redundancy. Redundancy is necessary because PCs are becoming increasingly integrated into industrial automation processes and manufacturing execution systems. Failures can have significant financial consequences - even though the Windows NT based PCs are not being used for direct control in hazardous or tight deadline closed loop applications. Although many researchers have investigated developing highly available Windows NT applications [5-9], none has addressed the issue in the context of process monitoring and control applications and the fault tolerance for distributed objects like DCOM. It is the design objective of the OFTT to provide application developers with a reusable, easily-integrated, OPC-33compliant middleware toolkit such that applications can be made fault tolerant with minimal modifications.

The remainder of the paper is organized as follows. Section 2 describes the design of the OFTT software architecture and its fault tolerance techniques. Section 3 presents the implementation experiences. An example is given in Section 4 to illustrate the application of the OFTT toolkit. Section 5 draws the conclusion.

2. OFTT Description

This section provides a top level description of the OFTT technology. The first subsection describes reference system configurations; the second identifies the software architecture and major components.

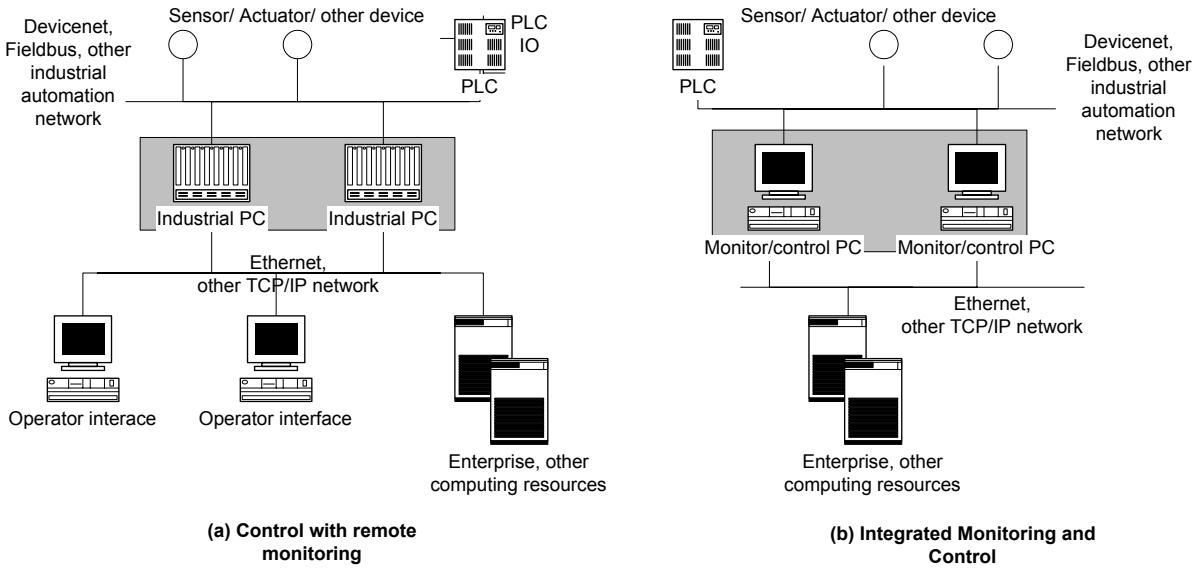

Figure 1. Reference System Configurations

2.1 System Configuration

OFTT is based on the *primary/backup approach*. As shown in Figure 1, two redundant computers are paired up via one or dual Ethernet networks and form a single logic execution unit. One is the *primary* node and the other the *backup* node. The same copy of an application (either an OPC server, or an OPC client, or both) resides on each node.

During normal operation, only the copy on the primary node is executed. In the meantime, its state is checkpointed and sent to the backup node periodically. In case of the primary fails due to either node failures, NT crashes, or application failures, the copy on the backup node will start running with the latest checkpoint.

2.2 Software Architecture

The basic design philosophy of the OFTT toolkit is to minimize the interference caused by adding fault tolerance on the normal application development process. A developer who has the domain-specific expertise yet has little background in fault tolerance can just focus on performance and functionality optimization in the design and implementation. If fault tolerance is also required, the application may simply add the services provided by the OFTT toolkit. This can be done at different levels of transparency, either by including a header file, inserting a single line in the application source code, or more sophisticated usage of the OFTT toolkit.

The OFTT toolkit is built on top of the Microsoft COM component architecture. Fault tolerance functions such as state checkpointing, failure detection and recovery are implemented as COM objects. Its top-level software architecture is shown in Figure 2. It consists of *OFTT engine*, *fault tolerance interface module (FTIM)*, *message diverter*, and *system monitor*. Details of each component are discussed in the following sections.

2.2.1 OFTT Engine

The OFTT engine is the core of the OFTT toolkit and controls all aspects of fault tolerance. In particular, it performs the follow functions:

- **Role management**: it determines the role of a node in the primary/backup pair (i.e., whether it is the primary or the backup) during the startup and switchover by negotiating with the peer node.

- **Failure detection**: it monitors the status of all software components that are linked with the fault tolerance interface module on the same

node and the status of the peer node by checking the heartbeat messages from each monitored component. If it does not receive the message after the pre-specified timeout, it considers the component fails and initiates a recovery provision. Failure detection for itself is done by the OFTT engine on the peer node. It simply sends out the heartbeat message periodically.

- **Recovery management**: How to recovery from a detected failure is controlled by the *recovery rule* that specifies whether to initiate a local recovery (e.g., a transient fault), or to transfer control to the backup node (e.g., a permanent fault). An application that uses the OFTT can explicitly specify the recovery rule either statically at compilation time or dynamically at run-time. The current implementation only supports static decision.
- **Status reporting**: it reports and updates the status of each monitored component to the system monitor.

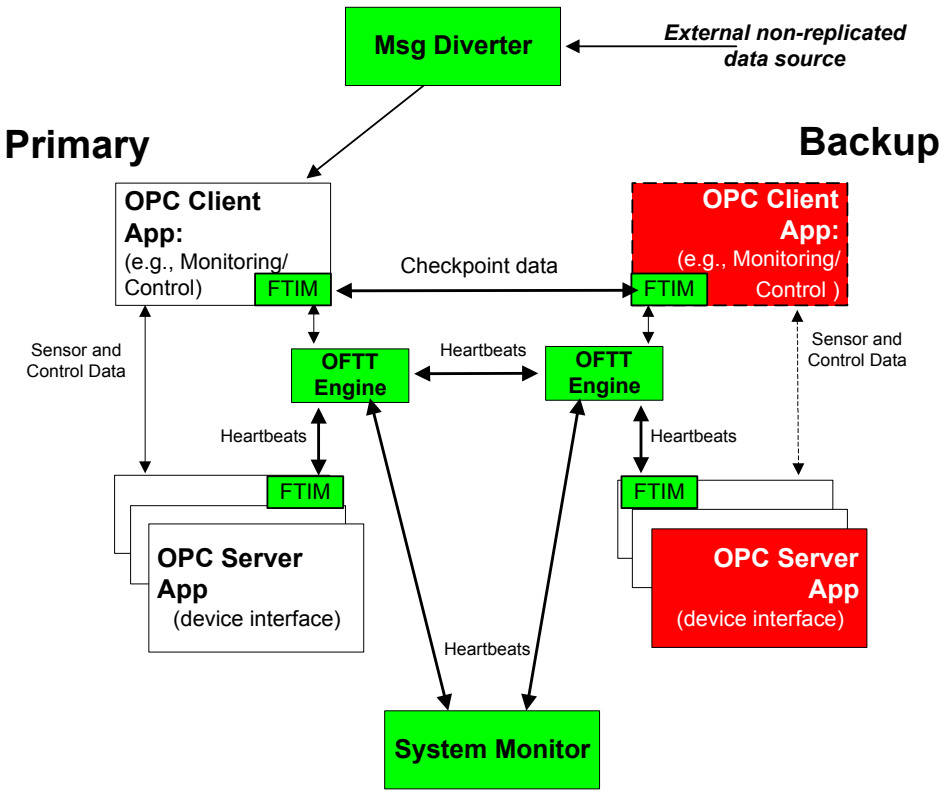

Figure 2. OFTT Software Architecture

OFTT engine is implemented as a client-side COM server and runs as a separate process started by the application.

2.2.2 Fault Tolerance Interface Module (FTIM)

Fault tolerance interface modules are responsible for checkpointing the application state, monitoring the status of the application, and communicating with the OFTT engine. It is implemented as a client-side COM server in the form of dynamic link library (DLL) and is linked to an application that wants to use OFTT services at the compilation time.

As mentioned previously, the OPC specification identifies two different types of applications, OPC server and OPC client. An OPC server is simply responsible for converting data from different types of I/O devices into the standard format. In this aspect, it is *stateless*. In contrast, an OPC client usually performs more sophisticated functions and uses data from many OPC servers. As a result, to reduce the overhead incurred by checkpointing the state, two different types of fault tolerance interface module are

provided: *OPC client FTIM* and *OPC server FTIM*. The difference between these two is that OPC client FTIM takes checkpoints while OPC server FTIM does not.

State Checkpointing. In the OFTT design, the application and the FTIM run as two separate threads within the same address space. The main application thread performs the task and may contain multiple threads. On the other hand, the FTIM thread, incorporated as a DLL, is responsible for taking checkpoint (for OPC clients), sending the heartbeat messages to the OFTT engine, and receiving the control from the OFTT engine. For statically generated kernel-objects such as threads, its context can be easily obtained using the standard Win32 API (*GetThreadContext ()*) and a memory walkthrough will extract the relevant data such as stack, global variables. For some dynamically generated kernel-objects such as a thread generated dynamically by the main application thread (by using *CreateThread()*), its handle (the starting address) can not be accessed directly through the standard Win32 APIs. In this case, a mechanism that manipulates the IAT (Import Address Table) and intercepts corresponding Win32 APIs has been developed to obtain the information.

Application Programming Interfaces (APIs). As stated before, the OFTT toolkit provides the application with a set of APIs in order for it to add fault tolerance. As a result, it is not totally transparent compared to the method in [9]. However, as has been shown in [10,11], in some cases, user directed checkpointing mechanism can improve the performance. Moreover, some event-based checkpointing is necessary. As a result, the OFTT has developed a set of APIs that allows the application to use the fault tolerance in different levels of transparency. The following basic set of APIs has been provided:

- *OFTTInitialize()*: the application requires the OFTT services. At the minimum, it is the only API an application needs to add in order to use the OFTT services.
- *OFTTSelSave()*: Checkpoint variable designation. It identifies specific variables that need to be checkpointed.
- *OFTTSave()*: Checkpoint save. Copy the address space (or the selected subset) and the stack to the peer node immediately, without waiting for a checkpoint period.
- *OFTTGetMyRole()*: Identify role (primary or backup) of a node
- *OFTTWatchdogCreate(), OFTTWatchdogSet(), OFTTWatchdogReset(),OFTTWatchdogDelete()*: Used to manage a reliable watchdog timer object.
- *OFTTDistress()*: Report a significant problem in the application to the OFTT engine and request a switchover (if application on the peer node is functional).

2.2.3 Message Diverter

The Message Diverter allows the primary/backup nodes to be a consistent logic unit that interacts with other applications and handles all I/O messages to and from applications, and diverts messages to the correct node. The current implementation uses Microsoft Message Queue. In particular, the message queue will store and transmit messages to the primary copy of the application. If a message is sent during a switchover, the message non-delivery is detected and retried.

2.2.4 System Monitor

The System Monitor displays the status of the components in a process monitoring and control system including hardware, operating system, OFTT components, and applications. Although necessary for system test, evaluation, and maintenance purposes, it does not need to be present for the operation of the OFTT fault tolerance provisions.

3. Implementation Experience

The work presented in the paper has shown that component-based software architecture such as COM is a viable means to support fault tolerance in the development of industrial process monitoring and control applications. In the meantime, the following development experience has been observed.

3.1. Access Information of Kernel-Related Objects

While Windows NT provides a rich set of Win32 APIs and libraries, it also complicates the programming effort, especially those related to kernels. The information on dynamically created kernels such as threads is not directly accessible via standard APIs. This creates difficulties in implementing checkpointing and recovery mechanisms as stated before. In addition, there exist a significant amount of functions and features that are documented little or none at all. As an example, the performance monitor is claimed to be a powerful utility to access the NT kernels. However, it is not

completely specified and in some cases is just misleading. For instance, the thread start address in the performance counter is always the pointer to a routine in NTDLL.DLL [12] and thus can not be used as the start address of a thread created dynamically.

3.2. Non-determinism of Windows NT

The lack of determinism in Windows NT start-up and thread dispatching did result in a number of design changes in the checkpointing and recovery scheme. For instance, the start-up logic was originally designed as follows: when a node starts up, it is in backup role and waits for a periodic time stamp from its peer node. It will shut down itself if it does not receive the message after a time-out period. This logic was used to minimize the impact of network failures (i.e., both nodes becomes the primary). However, because of the lack of predictability in the start-up time, the first node that starts up would frequently shut down since the second node may not start operation of the OFTT middleware before the time-out period elapsed. As a result, additional logic was added to initiate retries several times before it shuts down. It effectively solves the original problem.

3.3. DCOM Issues

While DCOM is powerful, it does have several limitations that have hindered the development of the OFTT toolkit. First of all, the DCOM does not have a well-defined built-in fault tolerance infrastructure. For example, its RPC service does not behave well in the presence of failures, and additional design efforts have to be made in order to compensate for the deficiency. Second, generation and installation of the DCOM server object proxy and stub increase extra development and configuration management effort. Some bugs encountered were due to the complexity.

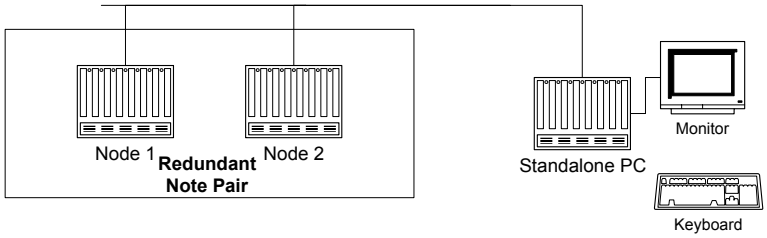

Figure 3. Demonstration Configuration

4. An Application Example

We have developed one application to demonstrate how the OFTT toolkit can be integrated to support the required fault tolerance. The application keeps track of the usage of a simulated small office telephone system that consists of 5 telephone lines and 10 callers. Numbers of busy lines are displayed in the histogram. The application is preferred to be fault tolerant since it records the past and present states of the system.

Figure 3 shows the hardware configuration. It consists of 3 PCs connected via an Ethernet. Two nodes are the primary/backup pair and run the application that has been added with the fault tolerance capability by the OFTT toolkit. The third PC acts as the user interface and test computer. The software configuration is shown in Table 1.

We will demonstrate the ability of the system to continue operating in the presence of the following failures:

a. node failure,
b. NT crash (blue screen of death),
c. application software failure,
d. OFTT Middleware failure.

Table 1. Software Configuration

Node	Software Element
Primary	OFTT Engine
	Call Track application (linked to OFTT Client FTIM)
Backup	OFTT Engine
	Call Track application (linked to OFTT Client FTIM)
Test and Interface	OFTT System Monitor
	Telephone System Simulator Calling History generator

5. Conclusion

OFTT is a middleware toolkit based on the COM architecture that can be easily integrated to process monitoring and control Windows NT applications where high-availability is of primary importance. It is also suitable for the large installed base of monitoring and control software running on Windows NT PCs. In addition to industrial applications, the OFTT toolkit can be used in other environments where high availability is a benefit. These include continuous environmental monitoring, laboratory automation, and multiparameter patient monitoring.

6. Acknowledgement

This work was sponsored in part by the NASA under the Small Business Innovative Research Program, Contract NAS 8 97037. Dr. Eltefaat Shokri's effort in the project is greatly appreciated. The authors also wish to thank anonymous reviewers for their inspiring comments and helpful suggestions.

References

[1] I. Breskin, "SCADA Takes the Factory Floor", *Managing Automation*, April, 2000. http://www.managingautomation.com.

[2] R. Daly, "Process Control and Execution", *Managing Automation*, June, 1999, pp. 34-36.

[3] OPC Foundation, "*OPC Overview*", v1.0, http://www.opcfoundation.org, Oct. 1998.

[4] Microsoft Corp. and DEC., "*The Component Object Model Specification*", Redmond, WA, 1995.

[5] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera, and J. Gray, "The Design and Architecture of the Microsoft Cluster Service", *Digest of FTCS-28*, pp. 422-431, June 1998.

[6] H. Abdel-Shafi E. Speight, and J. K. Bennett, "Efficient User-Level Thread Migration and Checkpointing on Windows NT Clusters", *Proc. Of the 3rd USENIX NT Symposium*, Seattle, Washington, July, 1999.

[7] J. Srouji, P. Schuster, M. Bach, and Y. Kuzmin, "A Transparent Checkpoint Facility On NT", *Proc. Of the 2nd UNENIX NT Symposium*, Seattle, Washington, pp. 77-85, August, 1998.

[8] Y. Huang, P. E. Chung, C. Kintala, C.-Y. Wang, and D.-R. Liang, "NT-SwiFT: Software Implemented Fault Tolerance on Windows NT", *Proc. Of 2nd USENIX Windows NT Symposium*, pp. 47-54. Seattle, Washington, August, 1998.

[9] P. E. Chung, W.-J. Lee, Y. Huang, D.-R. Liang, and C.-Y. Wang, "Winckp: a Transparent Checkpointing and Rollback Recovery Tool for Windows NT Applications", *Digest Of FTCS-29*, June, 1999.

[10] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix", *Proc. Of USENIX Winter Technical Conference*, New Orleans, LA, pp. 213-223, January, 1995.

[11] Y. Huang and C. Kintala, "Software Fault Tolerance in the Application Layer". In M. Lyu, editor, *Software Fault Tolerance*, pp. 231-248. John Wiley & Sons, 1995.

[12] M. Pietrek, private communication, August, 1999.