C-020

ソフトウェアによる制御用 FTC の実現に向けたオペレーティングシステム同期機能の開発
Synchronization Mechanism of Operating System towards realization of Software FTC for Control System

関山 友輝† Tomoki Sekiyama　吉田 雅徳† Masanori Yoshida　大島 訓† Satoshi Oshima　大平 崇博† Takahiro Ohira

1. はじめに

社会インフラを支える制御システムでは，短時間のサービス停止であっても大きな損失に繋がるため，高い信頼性・可用性が必要とされる．加えて，制御対象で発生した事象に対する応答のリアルタイム性も要求される．こうした制御システムを構成する計算機では，CPU やメモリ，ディスク，NIC といった構成部品を多重化し，その一部で障害が発生しても正常に動作を継続できるようにしたフォールト・トレラント・コンピュータ(FTC)が用いられる場合がある．この方式は，アクティブ・スタンバイ構成と比べて障害発生時にも状態の引き継ぎに必要なフェイルオーバー時間が極めて短時間で済むため，リアルタイム性の確保がしやすいという利点があり，制御システムに適している．一方で，CPU 等を冗長化した特殊なハードウェアが必要になるという課題があった．

本提案では制御システム向けの FTC を専用ハードウェアを使用せずに実現することを目的として，複数の一般的な PC サーバをネットワークで疎結合し，ソフトウェアによって FTC を実現するサーバ間同期手法を提案する．本研究では，各 PC サーバ上で動作するオペレーティングシステム(OS)内でサーバ間の同期をとることにより，その上でアプリケーションを改変せずに信頼性を向上させることを目指す．本報告では，Linux® 2.6.12 をベースとして提案手法によるサーバ間同期機能を実装し，評価を行った．

2. ソフトウェアによる FTC の構成

2.1 システムの構成

本提案では，ネットワークで接続された対象を制御するシステムを想定し，図 1に示すように，複数の PC サーバ，および Voter と呼ばれる多数決用計算機で FTC を構成する．図 1はサーバを 4 台構成とした例である．各サーバでは同一の制御アプリケーションを同期実行させ，障害発生時には当該サーバを FTC から切り離すことで可用性を向上させる[1]．

外部機器からのパケットは，Voter が受信したのち，各サーバにブロードキャストする．外部機器への制御パケットは，いったん各サーバから Voter に送信される．Voter は一定時間範囲内に到達したパケットを照査し，同一内容であることを確認したうえで，パケットを 1 つに集約して外部に送出する．パケット内容に相違があった場合，多数決により多数派となったものを選択し，少数派のサーバは FTC から切り離す．Voter 自身も多重化されており，相互に動作を照査しながら処理を進めることで，信頼性の向上を図っている．

なお，偶発的なパケットロストやネットワーク障害に備えるため，内部ネットワークおよび外部ネットワークは二重化し，サーバおよび Voter は両方のネットワークに同一内容を送信することにより，信頼性を向上させている．

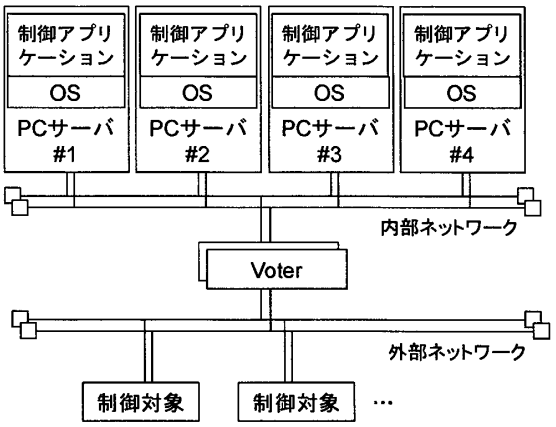

図1　ソフトウェア FTC のシステム構成

2.2 同期対象アプリケーションの前提

本提案で対象とする制御システムのアプリケーションは，以下を前提とする．

- 複数のプロセスから成り，それら全てが 1 つの CPU コア(同期コア)上で動作する．
- 各プロセスはシングルスレッドで動作する．
- プロセスは優先度に応じて CPU コアを割り当てられ，I/O 発行等でブロックされるか自発的にプロセスを切り替えない限り，ノンプリエンプティブで実行され続ける(リアルタイム・スケジューリング)．
- リアルタイム性を保証するため，システムコール等の処理時間が予測可能であることを必要とする．
- ページフォルトによる動作遅延を避けるため，各プロセスが使用するメモリ空間には，起動時に一括で物理メモリを割り当てる．
- 各プロセスにはその役割に応じた ID が振られており，これによって同一のプロセスであることを，サーバを跨いで一意に識別できる．
- 同期開始時点にはアプリケーションのプロセスは起動していない状態とする．また，各サーバのアプリ

† (株)日立製作所　Hitachi, Ltd.

ケーションが使用するデータは，同期開始前にサーバ間で一致させておくものとする．

2.3 同期対象プロセスと非同期プロセスの共存

サーバ上では，アプリケーションを構成するプロセス以外に，OS の機能を司るデーモンプロセスや，サーバを管理するためのプロセス等が動作する．これらは個々のサーバで異なる事象を扱う必要があるため同期対象とせず，アプリケーションのプロセスが使用しないコア(非同期コア)上で動作させる．このようなプロセスを非同期プロセスと呼ぶ．他方，アプリケーションのプロセスは同期対象プロセスと呼ぶ．

同期対象プロセスと非同期プロセスの間では，基本的には資源は共有せず，特に同期対象プロセスが使用するデータの更新は非同期プロセスから行わないものとする．ただし，アプリケーションの制御を行うため，サーバ間のタイミングのずれが一定時間である場合に限り，以下の操作を可能とする．

- プロセスの起動
- シグナルの送信
- メッセージ送受信(プロセス間通信)

非同期プロセスのほか，ハードウェア割込みの処理も非同期コアで行い，同期コアの動作を遅延させないようにする．Linux®では，TCP/IP や UDP/IP のプロトコルスタックの一部が割り込みハンドラとして実装されているため，本提案では実際のネットワーク通信は非同期コアに割り付けたリアルタイムプロセスから発行するものとし，このプロセスに同期対象プロセスとの間でプロセス間通信によりメッセージを仲介させる手法を採る．

3. オペレーティングシステムによるサーバ間同期手法

本章では，サーバ間でアプリケーションの動作を同期させるために，OS に加えた変更点について説明する．

アプリケーションの動作をサーバ間で一致させるためには，アプリケーションを構成するプロセスへの入力を全サーバで一致させるのに加えて，プロセスの実行順序を一致させる必要がある．これは，実行順序が異なると，プロセス間で共有する資源(共有メモリ，ファイル，プロセス間通信用バッファ等)へのアクセス順序が相違し，その内容が不一致となる可能性があるためである．以下では，プロセスの実行順序を一致させるために同期コアのプロセススケジューラに加えた変更を説明したのち，個々のプロセスへの入力となるシステムコールの実行結果を一致させる手法について説明する．

3.1 プロセススケジューラにおけるアプリケーションの同期実行

2.2節で述べたように，本提案では同期対象のプロセスはノンプリエンプティブにスケジューリングされる．このため，各サーバでプロセスの同一個所から実行を始めた場合，入力が同一であれば，次に別のプロセスに切り替わる契機(システムコール呼び出し等)は，全サーバで同一となることが保証される．したがって，各サーバで実行されるプロセスの順序を同一にするためには，同期コアのプロセススケジューラにおいてサーバ間で同一の次プロセスを選択すれば十分であると考えられる．

ただし，あるプロセスが I/O 完了等を待機していた場合，そのプロセスが実行可能になるタイミングはサーバによって異なるため，そのプロセスが全サーバで実行可能になるまで選択を遅延させる必要がある．一方，一部サーバで障害によりプロセスの実行が阻害された場合には，当該プロセスがいつまでも実行可能にならないことがありうる．障害が発生したサーバを検知して切り離すには，両者を識別する必要がある．このために本提案では，プロセスが選択されるまでの遅延時間に待機対象に応じた上限値を設け，これを超えても選択できないサーバを障害とみなす手法を採る．この方法は，待機対象に基づいて待機時間の上限が決定できるため，アプリケーションのリアルタイム性を保証する観点からも妥当であると言える．

次プロセスの選択ステップを以下に示す．

1. 同期コアにおいて，プロセススケジューラが呼び出されたら，その時点で実行可能状態となっているプロセスの ID の一覧を他のサーバに通知する．プロセス ID 一覧はプロセスの優先度に応じてソートしておく．また，実行可能状態となってから選択までの遅延時間の上限を超えたプロセス ID には，タイムアウトを示すフラグを併せて通知する．
2. 他サーバからの通知を受信する．一定時間内に同期中の半数未満のサーバから通知がなかった場合，同期が維持できていないと判断して当該サーバを同期対象から切り離し，他サーバにその旨を通知する．過半数のサーバから応答がない場合は，自サーバが他と異なる動作をしたと判断し，同期対象から自ら離脱するとともに，その旨を他サーバに通知する．なお，応答の有無が同数になった場合は，最も番号の小さいサーバを優先して同期対象に含める．
3. 受信した実行可能プロセスの一覧から，多数決で次プロセスを決定する．同期中の全サーバの一覧に同一のプロセス ID が含まれていれば，その中で一覧の先頭に最も近いものを選択する．全サーバが同じプロセス ID を選択できない場合，いずれかのサーバでタイムアウトしているプロセス ID があれば，その中から最も多くのサーバから提示されているプロセスを選択し，そのプロセスを実行できないサーバで障害発生したとみなして同期対象から切り離す．タイムアウトしているものがなければ，4 に進む．
4. idle プロセスを選択し，同期コアを待機状態にする．自サーバで新たにプロセスが実行可能になるか，一定時間が経過した場合，選択ステップを 1 から再実行する．

3.2 システムコール内におけるプロセス切り替え箇所の一致化

Linux®の I/O を行うシステムコールには，要求された I/O が完了するまでプロセスをブロックするものがある．これらのシステムコールでは，実際に I/O 要求を行う前に

必要な I/O が完了しているかを判定するケースがある．このとき，完了していない場合にはいったん別のプロセスに実行を切り替えるが，既に完了していた場合にはプロセス切り替えを行わない．

一例として，通常のディスク上のファイルを読み出す場合について説明する．プロセスがファイル読み出しのシステムコールを発行すると，カーネルはまず対象データがファイルキャッシュに存在するかを確認する．存在する場合には，ディスク I/O は発行せずにキャッシュ上のデータを直ちに返す．このときプロセスはブロックされず，プロセス切り替えは発生しない．しかし，キャッシュ上にデータがなかった場合には，実際にディスク I/O が発行される．このときプロセスはディスク I/O 完了までブロックされ，別のプロセスに切り替えられる．

その他の例として，同期対象プロセスと非同期プロセスとで共有される資源の排他処理が挙げられる，ファイルシステムのメタデータ等にアクセスを行う一部のシステムコールでは，内部でセマフォを用いた排他を行うものがある．セマフォが競合すると，先にセマフォを確保した側が解放するまで，後で確保しようとしたプロセスがブロックされる．非同期プロセスがセマフォを確保するタイミングはサーバ間で異なるため，こうしたシステムコールを同期対象プロセスが呼び出すと，サーバごとにプロセス切り替えの有無が相違する可能性がある．

このようにプロセス切り替えの有無が各サーバの資源の状態によって異なるシステムコールでは，プロセススケジューラの同期のみではアプリケーションの動作を一致させることができない．こうしたサーバ間で一致するとは限らないプロセス切り替えの条件を，以降では非同期条件と呼ぶ．

本提案では，非同期条件によりプロセス切り替えを行うシステムコールの実装を変更し，必ず全サーバでプロセス切り替えを行うようにするアプローチを採った．この変更が必要なシステムコールは数多くあるため，統一的な対処ができることが望ましい．そこで，非同期条件の判定を含む一連の処理をいったん非同期コアに委譲し，その処理が完了したのち同期コアに処理を戻す方法を採る．このとき，非同期コアにおいて同期対象プロセスの資源へのアクセス順序を一致させないと，サーバ間で不整合が発生する原因となるため，アクセス順序は委譲前に決定しておき，非同期コア上ではその順序に従ってアクセスを行う．

実際にシステムコールに変更を加えるには，図 2に示すように，非同期条件を含む処理の前後に，それぞれ非同期コアへの処理委譲と復帰を行う関数を挿入すればよい，関数には非同期コアからアクセスする資源を指定し．これをもとにアクセス順序の保証を行う．なお，システムコールによっては，終了に必要な条件が満たされるまでプロセス切り替えを繰り返しながら待機を行うものや，複数の非同期条件によって何度もプロセス切り替えを行うものもあるが，各判定のたびに実行コアを切り替える必要はない．複数の判定を一括して非同期コアに委譲することで，本手法による同期オーバーヘッドの低減を図ることができる．

```
  : // 同期実行
migrate_to_async_core(使用リソース); // 非同期コアへ委譲
if / while (非同期条件) {
   プロセス切り替え();
}
migrate_to_sync_core(使用リソース); // 同期コアへ復帰
  : // 同期実行
```

図2　システムコールの変更例

図 2の migrate_to_async_core()では，以下を実行する．

1. 指定されたリソースへのアクセス順を管理する単方向リストの末尾に自プロセスを追加する．
2. 非同期コアに自プロセスの割り付けを変更し，プロセス切り替えを発生させる．
3. 非同期コアでプロセスの実行が再開されたら，アクセス順を確認する．自プロセスがリストの先頭にない場合には，プロセスを待機する．

migrate_to_sync_core()では，以下の手順を実行する．

1. アクセス順のリストから自プロセスを除去する．
2. 新たにアクセス順の先頭になったプロセスがあれば，そのプロセスを起床させる．
3. 同期コアに自プロセスの割り付けを変更し，プロセス切り替えを行う．

同期コアでプロセスが再開されるタイミングはプロセススケジューラの同期により全サーバで同一となるため，これにより同期コア上で実行される処理内容が同一になることを保証することができる．

調査した 123 個のシステムコールのうち，おもにディスク上のファイルシステムへのアクセスを伴うものを中心に，55 個のシステムコールにこの手法が適用できた．

3.3 システムコール実行結果の一致化

前節で述べた手法は，非同期条件によるプロセス切り替えの有無には対応可能だが，非同期条件によってシステムコールの実行結果が変化する場合には，アプリケーションへの入力が異なることになるため対応できない．

アプリケーションの入力となるシステムコールの結果には，ディスクやプロセス間通信等から読み取ったデータと，成功またはエラーを示す返り値がある．

データのうち，ディスクや通信で得たデータの同一性をサーバ間で検証するには，サーバ間で大容量のデータに対して照合を行う必要がある．比較処理によって同期のためのオーバーヘッドが大きくなるのを避けるため，本提案ではデータの照合は行わないものとする．本提案のサーバ構成では，Voter による外部出力パケットの比較時に相違がないことでデータの整合性が確認できるため，システムコール内で照合を行わなくとも信頼性は低下しないと考えられる．

一方，システムコールのエラーは，サーバ間で常に同時に発生するものと，非同期に発生するものに分けられる．前者には，引数の誤りや同期対象プロセス間での通

信に関するものなどが含まれ，全サーバで同時に発生するため同期の維持には影響しない．後者は，以下に分類できる．

1. I/O エラー等，各サーバのデバイスの異常によるもの
2. メモリやディスク等のリソース不足
3. 同期対象プロセスと非同期プロセスの通信タイミングによるもの

上記のうち，1 の発生時には，発生したサーバを同期対象から切り離すべきである．2 に関しては，適切にアプリケーションのリソース設計を行うとともに，メモリ資源に関してはメモリ回収機能の強化[1]によって予防することが可能である．3 に関しては，同期を保証する必要があるため，サーバ間でタイミングの差異を吸収する機能が必要となる．このためには，分岐条件となる通信の有無をサーバ間で交換し，全サーバで同一の条件が揃うまで待機する必要がある．この際，待機時間は最大でも非同期プロセスの動作のずれの許容限界として設定した時間で十分である．

以下では，同期対象プロセスと非同期プロセスとの通信のうち，プロセス間通信によるメッセージの受信を例として説明する．プロセス間通信では select システムコール等を利用し，タイムアウトを指定してメッセージの到着までプロセスを待機させることができる．このとき，メッセージの到着タイミングによってプロセスの動作は図 3に示すように以下の 3 通りに分かれ得る．

1. システムコール実行開始時に既にメッセージが到着しており，直ちにシステムコールが正常終了する
2. システムコール実行開始時にはメッセージが到着しておらず，いったん待機したのち，メッセージの到着によってシステムコールが正常終了する
3. メッセージが指定時間内に到着せず，タイムアウトでシステムコールがエラーで終了する

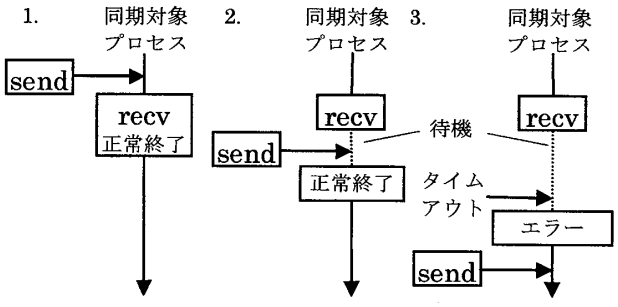

図3　受信タイミングによるシステムコールの挙動

このように待機の有無のみならず，システムコールがエラーとなるか否かが変化するため，前節の手法では動作が一致化できない．これに対処するためには，分岐条件であるメッセージの有無をサーバ間通信で交換し，相違があった場合は条件が揃うまで待機すればよい．ただし，一定時間以内に条件が揃わない場合には，少数側を挙動不一致とみなして切り離し，強制的に条件を一致させることで処理を続行する．

具体的には以下のステップを実行する．

1. サーバ間でメッセージの有無を交換する．ただし，一部のサーバから応答がない場合，3.1節ステップ2と同様に，少数となった側を同期対象から切り離す．

2-1. 全サーバでメッセージが有れば，直ちにシステムコールを正常終了させる．

2-2. 全サーバでメッセージが無ければ 3 に進む．

2-3. 一部のサーバのみでメッセージが有れば，1 に戻って再度メッセージの有無を確認する．ただし，一定時間これを繰り返しても全サーバでメッセージが揃わない場合，メッセージの有無に関して少数となった側を同期対象から切り離して処理を続行する．

3. メッセージが到着するかタイムアウトするまで，システムコール呼び出し元プロセスを待機させ，別プロセスに実行を切り替える．

4. 全サーバで待機中のプロセスが実行可能状態となると，プロセススケジューラにより同期してプロセスの実行が再開される．ここで再度 1 と同様にメッセージの到着有無を交換する．

5-1. 全サーバでメッセージが有れば，直ちにシステムコールを正常終了させる．

5-2. 全サーバでメッセージが無ければ，システムコールをタイムアウトによるエラーで終了させる．

5-3. 一部のサーバのみでメッセージが有れば，4 に戻って再度メッセージの有無を確認する．ただし，一定時間これを繰り返しても全サーバでメッセージが到着しない場合，メッセージの有無に関して少数となった側を同期対象から切り離して処理を続行する．

プロセス間通信のうち，UNIX ドメインソケットに関連するシステムコール 14 個を調査した結果，10 個に対して上述のような分岐条件の交換に基づく同期手法を適用する必要があった．

3.4 シグナルの同期

プロセス間通信の中でもシグナルは受信側プロセスの動作に割り込んで通知される点で特殊な対処が必要である．非同期プロセスから同期対象プロセスにシグナルを送信した場合，サーバによって送信タイミングが変化するため，同期を維持するには受信タイミングをサーバ間で一致させる手法が必要となる．

本提案では，同期対象プロセスがシグナルを受信可能なタイミングをシステムコール呼び出し時に限定することで，オペレーティングシステムのみの変更で同期がとれるようにした．

同期対象プロセスに対してシグナルが送信された場合，いったんシグナル送信を保留する．3.1節で述べたプロセススケジューラのサーバ間通信の際に，併せて保留中のシグナルの種類と送信先プロセス ID を交換する．全サー

バで同一の種別・プロセス ID のシグナルが揃ったら，保留していたシグナルを実際にプロセスに送信する．ただし，一定時間以内に揃わなかった場合には，多数決により少数となった側を同期対象から切り離す．

なお，同期対象プロセス間でシグナルの送受信を行った場合には，送信タイミングのずれはないため，上記の動作は行わずにシグナルを直接送信する．これは不正メモリアクセス等により，同期対象プロセス実行中に自プロセスに対して送信されるシグナルも同様である．

4. サーバ間同期手法の評価

前章で述べたサーバ間同期手法を Linux® 2.6.12 をベースに実装し，サーバ間の同期に必要となる処理時間のオーバーヘッド評価を行った．評価は，Xeon® 5140 (2.33GHz/4core)を搭載した 4 台のサーバを，2 重化した Gbit Ethernet で接続した環境において，アプリケーションの実行を全サーバ間で同期させた状態で行った．

4.1 停止時間の評価

本提案による同期手法では，一部のサーバで異常が発生した際，そのサーバを切り離して動作を回復するまでに要するフェイルオーバー時間は，各同期手法において設定されるタイムアウト時間によって決定される．

評価環境ではプロセススケジューラやシステムコール内のサーバ間通信時間のタイムアウトを 20ms として評価を行ったが，アプリケーションの同期実行に問題は発生しないことを確認できた．このことから，本手法による障害検知時のフェイルオーバー時間は 20ms 程度であると言うことができる．

4.2 プロセススケジューラのオーバーヘッド評価

プロセススケジューラによる同期のオーバーヘッドを計測するため，プロセススケジューラの呼び出しから終了までに要する時間を記録する処理を挿入した．その状態でプロセス切り替えを繰り返す評価用アプリケーションを実行し，1 回当たりの所要時間を記録した．記録されたプロセススケジューラの所要時間の分布を図 4に示す．

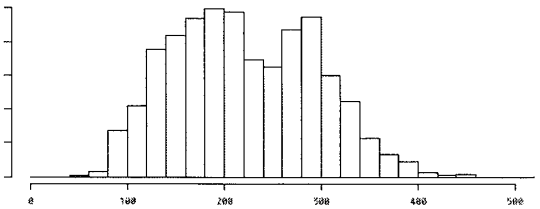

図4　プロセススケジューラの所要時間の分布 (μs)

この結果から，一回のプロセス切り替えに平均 200μs 程度を要していることが分かる．処理時間の内訳を調査した結果，ほとんどの時間がサーバ間通信の送受信のために使用されていた．従って，サーバ間の接続として現状の Ethernet よりも低いレイテンシを持つ Infiniband 等のインターフェースを利用することで，オーバーヘッドの低減が可能であると考えられる．

4.3 システムコール実行のオーバーヘッド評価

システムコール内に導入した同期手法による処理時間のオーバーヘッドを評価するため，対象システムコールを繰り返し実行する評価用アプリケーションを実行し，システムコールの 1 回あたりの平均実行時間を計測した．対象としたシステムコールと利用している同期手法，および計測結果を表 1に示す．

なお，表中の open システムコールおよび close システムコールは，ディスク上のファイルの open および close を繰り返し実行して計測を行った．ファイルシステムへのアクセスを伴うため，非同期コアとセマフォによる排他が必要となるため，これによるプロセス切り替えの有無がサーバ間で不一致とならないよう 3.2 節で述べた対処を行っている．加えて，open システムコールではファイルパスの検索に伴ってディスクアクセスを行う際のキャッシュヒットの有無によってもプロセス切り替えの有無が変化するため，併せて対処している．

また，recv システムコールでは，UNIX ドメインソケットにより非同期プロセスから送信したメッセージを繰り返し受信して計測を行った．この際，3.3節で述べた送信タイミングの有無がサーバ間で一致化される．send システムコールも同様に UNIX ドメインソケットにより非同期プロセスに向けてメッセージを繰り返し送信して計測を行った．この際には，送信元 UNIX ドメインソケット側に相手が未受信のパケットが一定以上蓄積されていないか，および送信先 UNIX ドメインソケットが受信可能な状態かの 2 つの条件によりプロセスの挙動やエラー内容が変化するため，これらが recv システムコールと同様の手法を 2 箇所で使用して一致化される．

表 1　システムコール実行時間の評価結果

システムコール名	同期手法	平均実行時間 (μs)
open	プロセス切り替え箇所の一致化(3.2節)	690
close	プロセス切り替え箇所の一致化(3.2節)	680
recv	実行結果の一致化 (3.3節)	104
send	実行結果の一致化 (3.3節) ×2 箇所	217

open システムコールおよび close システムコールの実行時間の内訳を調べたところ，プロセススケジューラの同期による時間が主たる所要時間を占めていた．これは，いったん idle プロセスに切り替わったのちに元のプロセスに戻るために，プロセススケジューラのオーバーヘッドが 2 回分かかるためである．このほかに，システムコールを実行するコアが 2 回移動することによるオーバーヘッドが 20～30μs 程度含まれていたが，現状では通信によるオーバーヘッドと比べて無視できる程度と言える．

recv システムコールおよび send システムコールでは，サーバ間で状態を交換するための通信に，1 回あたり 100μs 程度を要しており，システムコール実行時間の大半を占めていた．なおプロセススケジューラで発生する通信

時間と比較すると約 1/2 の時間となっているが，これらのシステムコールではサーバ間で交換する情報のサイズがプロセススケジューラよりも小さいためである．

プロセススケジューラと同様に，システムコール実行時間のオーバーヘッドに関しても，サーバ間をより低レイテンシのインターフェースで接続することで改善できると考えられる．

5. おわりに

本報告では，制御システム向けに FTC を複数の一般的な PC サーバとソフトウェアで実現するため，オペレーティングシステム内でサーバ間の動作を同期させる手法を提案し，また Linux®上に実装して評価を行った．

提案手法では，Linux®のプロセススケジューラおよびシステムコールの実装に変更を加えて同期機能を追加し，同期対象のアプリケーションを構成するプロセスの実行順序や入力をサーバ間で一致させることにより，サーバ間のアプリケーションの挙動を同期させることができた．また，障害発生時のフェイルオーバー時間を 20ms 程度に抑えることができ，制御システムに必要なリアルタイム性を確保することができた．ただし，オーバーヘッド評価の結果，サーバ間での通信時間により同期オーバーヘッドが大きくなっていた．今後，Infiniband 等のレイテンシの低いインターフェースでサーバ間を接続することで，よりオーバーヘッドを削減できると考えられる．

本研究は同期対象を制御システム向けのアプリケーションに限定しているため，スケジューリング方式としてリアルタイム・スケジューリングのみを対象としているなどの制約をアプリケーションに課している．これらの制約を緩和し，より広範なアプリケーションに適用可能にすることは今後の課題である．

参考文献

[1]大島 訓, 他, “ソフトウェアによる制御システム向け Fault Tolerant Computer の提案”, FIT2010 第 9 回情報科学技術フォーラム (2010).

Linux®は、Linus Torvalds の商標です。
Xeon®は、Intel Corporation の商標です。
その他記載の会社名、製品名はそれぞれの会社の商標もしくは登録商標です。