Vol.2019-HPC-168 No.2
2019/3/5

通信障害耐性を持った分散協調型実時間シミュレーション環境の構築

藤田 侑弥[1]　福間 慎治[1]　森 眞一郎[1]

概要：我々は, 遠隔地の高性能サーバで実行中のシミュレーションを複数の遠隔クライアントで共有し, 対話的な遠隔ステアリングが可能な分散協調型実時間シミュレーションのためのフレームワークを開発してきた. このような分散型システムでは, 特定のクライアントあるいは通信経路上で発生した障害がシステム全体に波及しない機能, ならびに, 障害復旧後にシミュレーションステアリングに復帰する機能が必要である. 本論文では, 我々の分散協調型シミュレーションのためのフレームワークにおいて, 広域網での通信障害に注目した障害対策について, 障害の検知, 障害発生時のシステムの持続性確保, 障害からの復旧後の対応について検討を行い, 通信障害耐性をもったフレームワークのプロトタイプを実装した結果を報告する.

1. はじめに

近年, ネットワークを通して遠隔地にあるコンピュータを用いるクラウドコンピューティングや, ユーザの近くにサーバを設置しサーバでも処理を行うエッジコンピューティングといったサービスが注目されている. 我々は, このようなサービスを応用とし大規模なシミュレーションを遠隔地の高性能な計算機上で実行し, ネットワークを通じてステアリング行い, リアルタイムに結果を観測できるシミュレーション環境の構築を目指して研究を行っている. その一つとして分散協調型シミュレーションのためのフレームワーク開発を行ってきた.

本研究の分散協調型シミュレーション環境とは, 遠隔地のシミュレーションサーバで行うシミュレーションと通信遅延の少ない操作端末の近くで行うシミュレーションが結果を連携させ行うシミュレーションである. このような分散システム環境下において問題となってくるのが通信遅延や通信障害である. 分散システムにおいて, システムの耐障害性を高めることは重要な課題の一つで特に自然災害への対策に関心が高まっており, 信頼性を保証することは必要不可欠なものとなっている. 我々は, 通信遅延に対しシミュレーションキャッシング [2] という 2 つのシミュレーション結果を連携させる技術を利用し通信遅延を隠蔽する研究を行って来た. しかし本フレームワークにおいて障害が発生した場合に各サーバでの対応, 通信障害時における動作等の通信障害耐性は未検討であった.

[1] 福井大学
Uniersity of Fukui

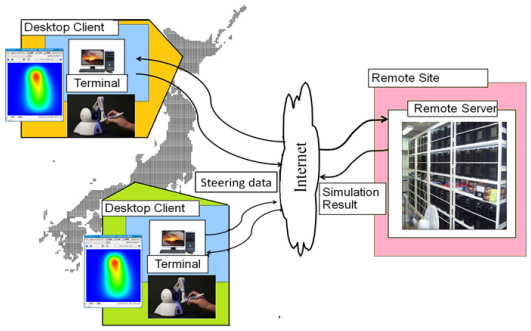

図 1　分散協調型実時間シミュレーション

本稿では我々の分散協調型シミュレーションのためのフレームワーク [3][4] に対し, 通信障害耐性を向上をさせた持続可能なシステムの開発を目的とする. 特に「通信障害」を広域通信網における通信遅延の著しい増加ならびにメッセージの消失として扱い一定時間内に復旧する可能性がある状況を考える.

2. 関連研究

大規模な計算を行うシミュレーションにおいては実行時間が長時間にも及ぶ場合がある. 障害発生時に最初から実行を行うのは望ましくなく機能を縮退させても動作し続けることが重要である. 広く使われている既存の故障対策としてチェックポイントとリスタートを用いたものがある. これは定期的にアプリケーションのスナップショットを保存しておき, 障害発生時にはそのスナップショットを用いて再度実行を行うといったものである. しかし, この手法では保存するスナップショットが膨大となることや障害時に

は再度実行する必要があること, アプリケーションのコードをシステムに合わせて記述するといった問題点がある.

また MPI の耐障害性に関連する研究は多く, 酒井ら [6] は障害の種類について MPI のライブラリを用いた正確な障害検知を行い, 様々な障害モデルについての事柄を示したシステムを提案している. 實本ら [7] は環境に合わせたフォルト/リカバリモデルについての MPI を提案し, 実装の評価を行っている. 吉永ら [9] は実際のアプリケーションに対し予備ノードを用いて実行継続手法についての評価を行っている. 彼らの多くは, クラスタなどの一つのシステム内での通信障害やノード故障における耐障害性である.

これらの研究に対し, 本論文では通信障害のみを対象としているが広域通信網との分散環境下における耐障害性について障害時の対応, 復旧の対応について検討を行う.

3. 分散協調型実時間シミュレーション環境

3.1 分散協調型実時間シミュレーション

手元にある計算機では実行不可能な大規模なシミュレーションを, 遠隔地にある高性能な計算機サーバ (リモートサーバ) で実行し計算条件や結果を通信する. この時, ネットワークを介して通信を行う際に通信遅延やジッタが大きな問題となる. そのため操作端末の近くにサーバ (ローカルサーバ) を用意しリモートサーバで実行中のシミュレーションの一部をローカルサーバ上でも並列に冗長実行する. リモートサーバで計算している結果を一定間隔で掲示できない場合にローカルの結果を表示する. これによってリモートサーバの通信遅延による影響を隠蔽し一定時間でのシミュレーション結果表示が可能となる.

ローカルサーバでの冗長実行には様々な形態 [3] が考えられるがここでは, リモートサーバで計算するシミュレーション領域よりも粗い解像度でシミュレーションを行う事を想定する.

3.2 一貫性制御手法

リモートサーバとローカルサーバで冗長に実行する際, シミュレーションが時間的な依存関係を持つ場合にシミュレーションが進むに連れ各サーバでの計算結果の違いが大きくなるという問題が発生する. そのため, シミュレーションがある程度経過した時点でシミュレーション内容をもう片方のサーバに送信する. これを一貫性制御と呼び, リモートサーバで計算している高解像度なシミュレーションデータをローカルサーバへと送り自身のシミュレーションに反映する. これによって, シミュレーション内容の誤差軽減だけでなく, ローカルサーバ側でも一時的に高精度なシミュレーションを行うことができる.

3.3 フレームワーク

上記の手法を実際のシミュレーションに対して適用する場合に, 自身のシミュレーションコードに対して追加や変更を行う必要がある. 本来のシミュレーションとは関係のない処理についての知識取得や実装を行うことは開発効率の低下や簡便性から好ましくない. そこで, 我々は上記の環境を実現するため必要な機能をフーレムワークとして実装した.

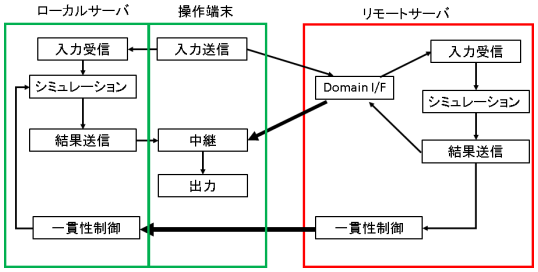

図 2 フレームワークにおける処理の流れ

3.3.1 入力送信プロセス

入力送信プロセスでは各サーバへ送信する計算条件の準備を行い送信を行う. フレームワークでは一定間隔での送信機能と一貫性制御時に, 一貫性制御が終わるまで送信を止める機能を関数として提供する.

3.3.2 Domain I/F プロセス

リモートサーバとローカルサーバとの中継を行うプロセスであり,Domain I/F とリモートサーバとの通信には Socket を用いて異なる MPI 通信グループに属するサーバとの通信を可能としている [3]. このプロセスでは操作端末からの入力データ受信およびリモートのシミュレーション結果を画像として操作端末へと送信する.

3.3.3 中継プロセス

中継プロセスでは各サーバからの計算結果受信, 一定間隔で出力プロセスへの送信を行う. このプロセスにより, リモートサーバから規定時間内にデータが来ていない場合でもローカルサーバの結果を送信することにより一定間隔でのデータ出力を保証している.

3.3.4 一貫性制御プロセス

一貫性制御プロセスでは一貫性制御のタイミングでリモートサーバのシミュレーションデータの送信を行う. 一貫性制御はシミュレーションや環境によって行う頻度を決定する必要があるため, ユーザが簡単に設定できる仕様として提供している.

4. 通信障害の影響分析

前述の通り, これまで開発してきたフレームワークでは, 実時間操作性を確保するという観点から, 数 10ms(国内) から数 100ms(海外) 程度の通信遅延とその揺らぎへの耐性を実装してきた. 将来復旧する可能性がある通信障害を, 通信遅延の著しい増大とみなせば, これまでの実装の延長とみなす事も不可能ではない. しかしながら, 数 10 分を超えるような通信遅延をミリ秒オーダの通信遅延と同じ枠組みで

考えるのは非効率的である. また, あまりにも長時間の通信遅延は OS レベルでの副作用（プログラムの強制終了）が発生し障害発生時の持続性確保の点で問題がある.

そこで, 実際に稼働中のシステムにおいて, ネットワークを意図的に遮断し, 大幅な通信遅延が発生した際の OS を含むシステム全体の挙動を調査し, 通信障害の影響を分析した.

まず, 何も障害対策を施さない状況でネットワークを遮断すると, その影響がリモートサーバならびに他のローカルサーバ（クライアント）に波及しシミュレーションの実行が停止した. また, 通信障害によりリモートサーバとの通信に障害が発生したローカルサーバでは, 一定時間後に MPI のコネクションタイムアウトが発生しローカルサーバ上の同一 MPI 通信グループに属すプログラムが強制終了された. 以下では, それぞれの事象が発生した際のシステム挙動の分析結果を示す.

4.1 リモートサーバ

リモートサーバでは操作端末からの入力を受信しシミュレーションを行い, 可視化データを操作端末へと送信する. しかし, リモートサーバの入力受信において操作端末と通信障害が起きているにも関わらず入力受信待ちとなっていた. これは, 従来のリモートサーバの実装方針が毎ステップ必ず受信データが来てシミュレーションを行う仕様のもと構築されていたためであった. そのため, 通信障害が発生した場合にリモートサーバは受信待ちとなり, 入力データが来るまで次の処理へと進まない.

4.2 TCP アルゴリズム

MPI はソケットと TCP を利用して実装されている. TCP には再送機能があり一定時間応答が確認できない場合には再びデータを送信し, より長い時間応答を待ち決まった回数の再送を行った後に送信を諦めるアルゴリズムがある. 想定する環境である Linux においてはデフォルトでこの時間が約 15 分に設定されている. この影響が次節で述べる MPD に与え, 作成したコミニュケータ内で通信が出来ないプロセスはコミュニケータから外されてしまう.

4.3 MPD(Multi-Purpose Daemon)

本研究で用いている MPICH を用いた MPI では事前にコミュニケータと呼ばれる通信を行うための集団を作る必要があり,MPD(Multi-Purpose Daemon) と呼ぶデーモンを実行する. 実行すると各ノードでリング状に接続され各デーモンは定期的に通信を行いコミュニケータの状態を保持する. コミュニケータ内において通信障害が発生した場合, 一時的にコミュニケータ内からそのプロセスは外れ通信回復後再びコミュニケータ内に戻る. しかし一定時間通信を試み回復出来なかった場合や, あるノードのデーモンが kill された場合は再度リングが結成され障害が発生したプロセスは外される.

4.4 MPI 通信

HPC 分野において並列化の際に MPI を用いた科学技術計算が多く使われている. 本フレームワークにおいても各プロセス間との通信には MPI を使用している. フレームワーク内で通信障害時の際に影響を受けるのが各サーバとの MPI 通信部分である. 通信障害の際に相手との通信が出来ない状態で通信ライブラリを呼び出すと MPI 関数から戻らず待ち状態になる. さらに一定時間経過すると MPI の標準設定ではエラーを返して実行中のプロセスを終了させるといった問題が起こる.

4.5 フレームワークへの影響

本フレームワークにおける一連の処理としては, 入力データの送信, シミュレーション計算, 計算結果送信, 一貫性制御処理, 結果出力となる. 前節での MPI 通信の影響がフレームワークに関係するプロセスおよび処理が,

(1) 入力送信プロセスと Domain I/F
　各サーバへと入力データを送信する.
(2) Domain I/F と中継プロセス
　リモートサーバでのシミュレーション結果を可視化して画像として送信する
(3) 各サーバの一貫性制御
　リモートサーバで高精度なシミュレーションデータを送信する

であり, 通信障害時にはこれらの処理を行うことが出来ず, 各プロセスの MPI 関数で送受信が終わるまで待ち状態となっていた.

5. 耐故障性を保証する機能設計

5.1 要求定義

我々が想定しているステアリングが可能なシミュレーションは, シミュレーション結果をユーザが直接体感しその反応を対話的に行うシミュレーションである. シミュレーションは入力に対し実時間で結果を掲示し続けることが求められている.

そのため本稿では, 実時間性をより重視しシステムによる中断が許されず一時的な機能の縮退を許容した上でシミュレーションが持続可能な環境の設計を目標とする. ここでは障害が復旧する可能性を考慮し障害中の動作および復旧時の対応も必要であると考える. 復旧に関してはシミュレーション結果の復旧もあるがフレームワークとしての機能を復旧することとする. 出来る限りのシミュレーション結果を正常な状態に戻すような工夫は行うがシミュレーション結果が完全に正常な状態に戻ることを保証することではない.

よって障害時でも各サーバのシミュレーションが動作し, 復旧時にフレームワークとして復旧可能であることをもってして持続性を持ったシステムであるとする.

5.2 リモートサーバの自律性確保

これまでに, 複数のクライアントがシミュレーションとステアリングを行うためのマルチクライアント拡張を行ってきた. これは不特定多数のクライアントが任意のタイミングでリモートサーバに接続ができるように構成されたものである. 各クライアントの参入や退出による影響を他のクライアントへ波及しないよう設計され, リモートサーバは各クライアントからの入力に対し処理を行う. しかし障害耐性を持ったサーバ構築を行う際に, 既存の入力駆動型ではクライアントからの入力を待ち, 通信障害時の際に障害とは関係のない他のクライアントへ影響が及んでしまう. そのためクライアントからの入力処理があった場合に処理を行う時間駆動型のような設計が必要である.

5.3 対策検討

前章での通信障害の影響から MPD レベルでのコミュニケータを維持する (送信間隔をのばす) 対策を考える. これは通信障害時においても MPD で作成される状態を通信が行える状態に保つことである. しかし, コミュニケータが再構築された場合でも MPI レベルの単純な通信のみは可能であり MPI_Send や MPI_Recv といった関数を呼び出し通信を行うことが可能である. しかし,MPD が提供する様々な機能を活用するため出来る限り MPD プロセスの離脱を防ぐ処置が必要である.

次に MPI レベルでのタイムアウト問題がある. 通信障害発生時に通信関数を起動すると OS(ならびに MPI) で定義する長時間の待ち状態 (CentOS6 のデフォルトで 15 分程度) に入り, その間に通信が完了できないと,MPI のデフォルトのエラーハンドリグ処理によりプログラムが強制終了される. これらの問題への対策として,「通信障害を隠蔽する方法」と「エラーハンドラを定義する方法」が考えられる.

通信障害を隠蔽する方法とは, 通信障害の発生を検知した場合, 通信処理の代替処理を行う手法である. 具体的な代替処理については次章で検討するが, 送信データのバッファリグや受信データの予測などがその一例である. これにより,MPI 関数での長時間のブロッキングを回避する. 通信関数の挙動を替える必要があるため, フレームワーク内のプログラムの改変が必要となるが, ユーザプログラム自体の改変は不要である. また, 次項で述べるラッパ関数を用いる事で本質的に必要なプログラムの変更量を低減する.

エラーハンドラを用いる方法は, エラー発生時の処理を定義することで無条件な強制終了を回避し代替処理を行う方法である. この手法単独でも通信障害対策は可能であるが, 障害発生時にシステム定義の長時間の待ちが発生するため, 本研究で想定している実時間シミュレーションステアリングのように頻繁に通信が発生すると実用的ではない.

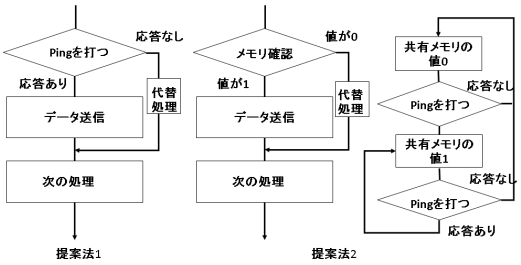

図 3 heartbeat による通信監視の流れ

そこで, 本システムでは「通信障害を隠蔽する方法」を基本とし, 障害検知に失敗して実行してしまった通信への対応として「エラーハンドラを用いる手法」を併用することで通信障害耐性を得ることとする.

5.4 実装

通信を隠蔽し通信を行わないようにするには, 通信の状態について監視する仕組みが必要であると考えられる. ここで多くのシステムで用いられている heartbeat 機能を参考に既存の MPI 関数に対しラッパ関数として実装する事を考える. 実装にあたり 2 つの手法が考えられ, 通信のライブラリを呼ぶ前に通信の状態を確認する方法 [提案法 1] と通信の状態を監視するようなプロセスを新たに設ける方法 [提案法 2] である.

提案法 1 が通信を行う際に ping を打ち通信の状態を確認する. 通信の応答があれば MPI の通信ライブラリを呼び, 応答がない場合は通信を行わず通信障害時の代替処理に移る. 提案法 2 では, 別のプロセスを用意しそこで ping を打ち続け通信の状態を常に監視し通信の状態を共有メモリに保存する.MPI の通信ライブラリを呼ぶ際にはメモリを確認して通信可能であれば通信を行い, 通信が出来ない状態であれば代替処理に移る.

提案法 1 では実際に通信を行いたい場合に ping を打ってから通信を行うため大きなオーバヘッドとなる可能性があるが, 提案法 2 ではメモリを確認し通信を行うため,1 よりもオーバヘッドは小さくなると考えられる. 一方, 障害検知率を考えると提案法 1 は通信の状態を確認してからすぐに通信を行うことができるが, 提案法 2 ではメモリを確認した時間の通信の状態がいつの通信状態であるか定かではないため提案法 1 のほうが障害検知率が高いと考えられる.

この関数は, フレームワーク内の各サーバ間とのノード間通信部分だけで使用すればよく, ユーザが各提案法を自身の環境に合わせて選択可能となっておりシミュレーションを変更する必要はない.

エラーハンドラを用いて制御を行う手法についてはすでに MPI が関数として用意されており,MPI のエラーハンド

ラを用いてエラーを返した時の処理を実装することで実現が可能である.

6. 障害時における代替処理

通信状態の検知が heartbeat によって可能となった. この機能により通信の状況を確認し通信障害時に通信待ち状態となることを回避することができる. しかし, 行うはずだった処理が通信障害時には行わないため本来想定していたシミュレーションに影響を及ぼす可能性がある. そこで, 通信障害時に行う代替処理をいくつかの状況に応じて検討を行う.

6.1 方針

4.2 節での要求定義より, 通信障害が復旧する可能性を考慮し復旧可能時には復旧できるシステムとし構築してきた. そこで, 障害時における代替処理を考える際に通信障害からの復帰を手助けすることが本システムにおいて求められている機能と想定した. そのため障害後のシミュレーションを保証する処理を代替処理の実装方針とする. その際, シミュレーション内容やユーザの環境によって, 通信障害時に行いたい処理および要求は異なると考えられる. そのため本稿では想定されるモデルに応じて代替処理を考え設計を行う.

6.2 想定する障害復旧モデル

本研究では, 以下にあげる 3 つの障害復旧モデルを設定し, それぞれの復旧モデルに対し, システムに求められる要件を定義する.

(1) 入力欠損無保証レベル

最低限の要件として, 特定のクライアントで発生した通信障害によりシステム全体のブラックアウト発生を回避するモデルである.

障害発生中は, 障害が発生したクライアントと, その他のシステムを分離し, それぞれが独立して自律動作するモデルである. 分離されたクライアントでは, 自らが取得した入力データのみでシミュレーションを継続する. シミュレーションの分解能 (あるいは空間サイズ) が低下する, 他のクライアントで発生した入力情報が反映できない等の機能縮退はあるが, シミュレーションの継続は可能である. 事後処理のため入力データを保存する処理は行わない. 一方, 障害発生ノード以外のシステムでは, 障害発生ノードからの入力データが反映できないという機能縮退はあるが, その点を除けば通常動作を行う. 観測データを入力としてシミュレーションを行うシステムにおける観測データ欠損時の動作がこれに対応する. 障害発生中は当該クライアントに対する一貫性制御動作は中断する.

障害から復旧した時点では特に何も処理を行うことなく, 次の一貫性制御のタイミングでリモートサーバの結果が障害発生ノードのシミュレーションに反映される.

(2) ベストエフォートレベル

(1) の機能に加え, 障害発生中ならびに復旧後のシミュレーション結果を可能な限り精度保証するモデルである.

障害発生中は, リモートサーバ, 障害発生ノード双方で入力データの欠損が発生する. そこで, 事後処理のために入力データのバッファリングを行うとともに, 現在進行中のシミュレーションに対しては, 入力に連続性が仮定できる場合等, 入力予測が可能であれば欠損した入力を補填してシミュレーションを実行する.

障害復旧後には, バッファリングしていた入力データを用いて必要な補正処理を行う. 具体的な処理についてはアプリケーションに依存するため, ユーザが指定することを原則とする. ユーザ指定が入力データの無視であれば (1) と同じ動作となる. シミュレーションキャッシングの応用モデルによっては, 障害ノードで継続実行して得られたシミュレーション結果が重要な情報を含んでいる場合も考えられる. そこで, このような情報をリモートサーバに反映することが求めらる場合も考えられる.

(3) 完全復旧レベル

このレベルの実装には, 非常に多くの計算資源 (演算性能, 記憶領域等) が必要であるが, これらが得られる場合に障害復旧の一定時間後に, 障害が無かった場合と同等のシミュレーション結果をリモートサーバで得るモデルである.

障害発生を検知すると, リモートサーバでは実行中のシミュレーション結果のスナップショットを取るとともに, それ以降に到着する全入力情報のバッファリングを開始する. 障害発生中の各サーバでの動作は基本的に (2) と同等である. 障害復旧時にはフォアグランドで (2) と同等の処理を継続するとともに, バックグランド処理として保存していたスナップショット, 障害サーバならびにリモートサーバに保存した時系列入力データを用いてシミュレーションの再実行を行う. クラウドサーバのスケールアウト機能等を用いてリモートサーバの計算性能を上げ,1 ステップのシミュレーションに要する時間がシミュレーション間隔よりも短縮でできる場合には, その差を利用してバックグランドで追従実行を行うと, 一定時間後には時間差がなくなり, その時点で障害が発生していない場合の真のシミュレーション結果が得られる.

この時点で入力データのバッファリング処理を停止するとともに, 保存していたスナップショットならびにフォアグランドで継続実行してきたシミュレーション

結果を廃棄する. 最新のシミュレーション結果は次の一貫性制御のタイミングで各クライアントに配信される. 理論的には復元が可能であるが, 障害発生期間が長い場合には現実的な時間での復旧は容易ではない.

6.3 提案する処理

上記の方法を実現するに当たり本論文では以下の処理を提案する.

6.3.1 予測

通信障害環境下においてはデータの送受信が行えず, 必要な情報が手に入らない. シミュレーションの内容によっては通信障害時においても可能な限り, 障害が起きなかった場合と同様な処理を行いたい (2) の方法を実現するための処理とし予測を行う. 情報の正確性をある程度許容する代わりに予測を行うことで通信障害時においても同様な処理を行う.

特に入力データに関して予測を行う場合を考える. 予測を行うに当たり様々な方法が挙げられるがここでは入力データを予測する際に, 一番シミュレーションへの影響が大きくならない (シミュレーションが想定していない入力値を予測値とし破綻等が起こらない) かつ信頼できる値でなければならないと考える. そこで, 障害前に受け取った最後のデータを予測値とする処理を提案する. 実際の入力データと異なる可能性が高くシミュレーションの精度は保証できないが, 予測という機能を用いて障害後でも入力のあるシミュレーションを行うことが重要であると考え結果の精度に関してはユーザに預けることとして本機能を提供する.

6.3.2 保存

通信障害からの復旧の際に, 通信障害中の情報入手や過去に戻ってシミュレーションをやり直したい場合が想定される. この時に通信障害中に送るはずだった情報が必要となる. そのため, 通信障害時にはデータを送信するのではなく保存しておくことで, 復旧時に利用することが可能となり適宜行いたい処理を選択できる. これは (3) の方法を実現するための処理として提供する.

この時問題となってくるのが, 保存にかかるメモリ量と復旧後保存したデータの処理方法である. 保存に関して, 障害からいつ復旧できるかは不明であることからシミュレーション結果や入力等のデータを全て保存することが理想であるが現実的ではない. そのため, 過去のデータ数回分をユーザがある程度定め古いデータ程圧縮し保存することとする. 一般的に古い情報よりも最新の情報のほうが需要があると考え, 入力データおよびシミュレーションデータを保存することを考える. 障害復旧時にこのデータを利用するかどうかはシミュレーションの内容に大きく関わってくるため, 復旧後どう扱うかはユーザに預けることとする.

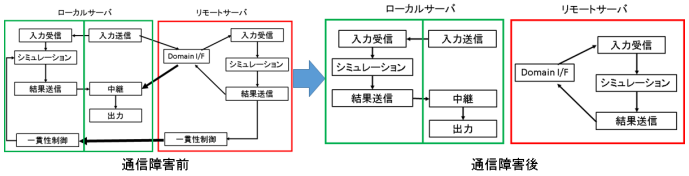

図 4 入力欠損無保証モデルの構成図

6.3.3 逆一貫性制御処理

逆一貫性制御処理とは, 本来リモートのデータを送りローカルのシミュレーションに反映する一貫性制御処理を, ローカルがデータを送りリモートのシミュレーションに反映するものである.

障害から復旧した直後であれば, リモートサーバは通信障害中の入力データを貰っていないのでリモートのシミュレーションは信頼性がない. しかしローカルのシミュレーションは入力データを貰い低解像度ではあるがシミュレーションを行っている. このとき信頼できるシミュレーション結果はローカルであると考えリモートにローカルのデータを送りリモートが自身のシミュレーションに反映する. リモートはローカルのデータを貰うことである程度信頼性を持ったシミュレーションが復旧後にも可能であること考えられる. 実際に我々の他の研究 [4] で実装しており, 本フレームワークの処理においても耐障害性のための処理として実装可能であると考える.

7. 実験と考察

7.1 検証

まず VPN を用いて福井と東京間でシミュレーションを実行させ, 意図的に通信が出来ない状態 (通信ケーブルを抜いた場合) でもシミュレーションを継続できるかおよびケーブルを元に戻した場合に復旧できるかを検証した. リモートサーバを東京, ローカルサーバを福井として入力欠損無保証レベル図 4 の状態で実験を行った.

結果, 多くの場合で通信障害時でもシミュレーションを続行させることができ, 復旧も可能であった.[*1] しかしケーブルを抜いた場合にいくつかのパターンでプログラムが止まってしまった. これは,heartbeat による通信状態確認と実際に通信を行う間で起きてしまう検知率の問題と一貫性制御のタイミングで各プロセスの送受信者どちらかが, 相手が通信処理を行うまで待つ通信待ち状態となってその後に通信が出来ない状態となって起きてしまう問題であると考えられる. 検知率の問題は, すべて提案法 1 にすることでより検知率を高めることができる. 通信待ちについては各プロセスがパイプライン処理で動いているため起きてしまう問題であり, 高速に処理を行うために必要な処理であり,

[*1] フレームワークの実装を容易にするため MPI の MPI_ANY_TAG を用いていたが不具合があり障害から 15 分経過後エラーが起きていた. ただし, 用いないことで対処がが可能である.

表 1 実験環境

	リモートサーバ (東京)	ローカルサーバ (福井)
CPU	Intel Xeon E5-2676	Intel Corei7 3770
OS	Linux4.9.27	Linux2.6.32
MPI	MPICH2-1.1	

表 2 各提案手法のオーバーヘッド

	提案法 1	提案法 2
時間 [ms]	70	0.002

避けられない通信障害である.

7.2 heartbeat 機能の適用

次にシミュレーションを実行させた際の heartbeat によるオーバヘッドを調査した. 提案法 1 だと実際に通信を行う際に 70 ミリ秒余計にかかってしまうが, 提案法 2 では 2 マイクロ秒でよく, 提案法 2 の時間は通信の環境によらずほぼこの時間であると考えられるため提案法 2 を実装した際の通信のオーバーヘッドは無視できると考えられる. しかし, 実験環境で行った場合に提案法 2 で通信状態を最後に確認した時間が最大で提案法 1 の時間となるため障害検知率は提案法 1 のほうが高い.

8. まとめと今後の課題

本研究では, 我々のフレームワークに対し通信障害の影響について各通信層で議論を行い, その対策とし heartbeat を用いた障害検知を実装した. また障害時における処理について検討を行い, いくつかの機能を提案した. 今後の課題としては, ここで提案している障害時の処理の実装があげられる. また, 障害時における処理については想定されるシミュレーションによって必要となる機能や種類が変わるためそれに対応した処理が必要である. そのため, シミュレーションに応じた処理についての検討および実装について進めていく予定である.

参考文献

[1] Ralph BUtler,William Gropp,Ewing Link :Components and Interfaces of a Process Management System for Parallel Programs:Workshop on Clusters and Computational Grids for Scientific Computing, Sept. 24-27, 2000
[2] 橋本健介, 手塚俊作, 森眞一郎, 富田眞治:シミュレーションキャッシングと遠隔インタラクティブ流体シミュレーションへの応用, 情報処理学会論文誌,Vol.5,No.4,pp.76-86(2012)
[3] 山本 優, 西村祐介, 福間慎治, 森眞一郎： シミュレーションキャッシングフレームワークの実装, 情報処理学会論文誌, Vol.57, No.3, pp.823-835(2016).
[4] Jiachao Zhang, Shinji Fukuma, Shin-ichiro Mori:Silk Road:A Framework for Distributed Collaborative Simulation, 情報処理学会論文誌,Vol59,No.3
[5] 堀田 勇樹, 田浦 健次朗, 近山 隆:耐故障並列計算を支援する自律的な故障検知機構, 情報処理学会論文誌,Vol.46,pp236-244(2005).
[6] 酒井 将人, 石川 裕:クラスタ監視機能付き MPI 通信ライブラリ, 研究報告ハイパフォーマンスコンピューティング (HPC),2005-HPC-103,pp.175-180
[7] 實本 英之, 松岡 聡:ポータブルな耐故障性コンポーネントフレームワークを持つ MPI 実装に向けて:研究報告ハイパフォーマンスコンピューティング (HPC),2004-HPC-101,pp.193-198
[8] 後藤田祥平, 柴田直樹, 山内由紀子, 伊藤実:クラウドコンピューティング環境でのマルチコアプロセッサの停止故障を考慮したタスクスケジューリング:マルチメディア, 分散協調とモバイルシンポジウム 2011 論文集,2011,1595-1603
[9] 吉永 一美, 亀山 豊久, 堀 敦史, 石川 裕:予備ノードを利用した故障後の実行継続手法の検討と評価:研究報告ハイパフォーマンスコンピューティング (HPC),2014-HPC-147(21),1-9