分散ストリーム処理システムにおける高信頼化手法の提案

塩川　浩昭† 　渡辺　陽介†† 　北川　博之††† 　川島　英之†††

† 筑波大学第三学群情報学類　〒 305–8573 茨城県つくば市天王台 1–1–1
†† 東京工業大学学術国際情報センター　〒 152–8552 東京都目黒区大岡山 2–12–1
††† 筑波大学大学院システム情報工学研究科　〒 305–8573 茨城県つくば市天王台 1–1–1
E-mail: †shion@kde.cs.tsukuba.ac.jp, ††watanabe@de.cs.titech.ac.jp,
†††{kitagawa,kawasima}@cs.tsukuba.ac.jp

あらまし　近年，センサデータなどのストリームデータに対する問合せ要求が増大し，それらを実現するストリーム処理システムが研究開発されてきた．そして，地理的に離れた情報源の統合や負荷分散を実現させるために，ストリーム処理システムを分散配置させて利用する分散ストリーム処理環境が注目されている．そのような分散環境では，中継ノードが停止することでシステム全体が停止してしまうという問題がある．本研究では，分散環境において高信頼化を実現する Lazy Standby 方式を提案する．本方式では，各分散ノード間の通信にバッチ処理を導入することで，既存手法である Active Standby 方式，Upstream Backup 方式を統一化し，リカバリ時間とバンド幅オーバヘッドの調節を可能にする．また本研究では，提案手法の評価実験を行い，本方式の動作特性を検証した．
キーワード　ストリームデータ処理システム， 高信頼化，分散コンピューティング

A High-Availability Method for Distributed Stream Processing System

Hiroaki SHIOKAWA†, Yosuke WATANABE††, Hiroyuki KITAGAWA†††, and Hideyuki KAWASHIMA†††

† College of Information Science,University of Tsukuba　Tennoudai 1–1–1, Tsukuba City, Ibaraki, 305–8573 Japan
†† Global Scientific Information and Computing Center, Tokyo Institute of Technology　Oh-okayama 2–12–1, Meguro-ku, Tokyo, 152–8552 Japan
††† Graduate School of Systems and Information Engineering, University of Tsukuba　Tennoudai 1–1–1, Tsukuba City, Ibaraki, 305–8573 Japan
E-mail: †shion@kde.cs.tsukuba.ac.jp, ††watanabe@de.cs.titech.ac.jp,
†††{kitagawa,kawasima}@cs.tsukuba.ac.jp

Abstract　Nowadays, a demand for query processing over stream data is increasing. Stream processing systems that fulfill continuous query processing for stream data have been developed. And distributed stream processing systems have been focused to use data sources which are located at remote sites, and to realize load sharing system. Since these systems must continuously monitor data from data sources, system failures bring serious damages to them. This paper shows a high-availability method which achieves dependability in distributed environment. Our method can generalize preceding methods Active Standby and Upstream Backup by using batch processing for communication between nodes. In addition, this paper shows the experimental results of our method.
Key words　Stream Data Processing System, High-Availability, Distributed Computing

1. ま え が き

近年，実世界センサデータや株取引きデータなどの絶えず情報を配信するストリーム型情報源が増大し，それらに対する継続的な監視等の処理要求が高まっている．例えば，次々と配信される株取引きデータを監視し，株価が一定値以上であったら通知する [16]，などは典型的な要求の一つである．このようなストリームデータに対する処理要求を実現するための基盤シス

テムとして，ストリーム処理システム [2]〜[4], [15] が注目されており，我々の研究グループも StreamSpinner [14] というストリーム処理システムを開発している．

一方，地理的に離れた場所から提供される情報源を扱う場合や，ストリームデータ処理を行う特定のマシンの負荷分散を行う場合には，ネットワーク上に分散した計算資源やネットワーク帯域・遅延を考慮した分散ストリーム処理システムを構築する必要がある．そのため近年では，分散ストリーム処理システム [1], [12] の研究が盛んに行われている．このような分散環境で，長期間ストリームデータに対する処理を行うには，各マシン上にあるストリーム処理システムが途中で停止することなく，安定動作し続けることが必要となる．そのため，分散環境においてシステムの高信頼化は重要な課題となっている．

現在，このような分散環境に対して，いくつかの高信頼化手法が提案されている [5]〜[9], [13]．これらの高信頼化手法の共通アイデアは，分散環境を構成する各マシン上のストリーム処理システムに対してセカンダリと呼ばれるバックアップ用のストリーム処理システムを別マシン上に用意し，あるマシンが故障した場合にはバックアップ用のマシンが故障を検知して問合せ処理を引き継ぐというものである．また，故障によるデータの消失を防ぐための機能など，データ整合性を維持するための仕組みを持つ．これにより，部分的な障害に対して問合せ処理が停止することなく，分散環境におけるシステムの高信頼化を実現している．これらの高信頼化手法は Hwang [6] らの研究により Passive Standby 方式，Active Standby 方式，Upstream Backup 方式に分類されている．

分散ストリーム処理システムにおける高信頼化手法の評価には，バンド幅オーバヘッド，リカバリ時間というトレードオフの関係にある 2 つの指標が使われる [6]．バンド幅オーバヘッドは高信頼化のために必要になるバンド幅，リカバリ時間は障害回復に要する時間のことである．各方式と指標の関係は次のようになる．Passive Standby 方式と Active Standby 方式ではバンド幅オーバヘッドが大きくなる一方，リカバリ時間が小さくなる．Upstream Backup 方式では，バンド幅オーバヘッドが小さくなる一方，リカバリ時間が大きくなる．即ちこれらの 3 方式では，バンド幅オーバヘッド，リカバリ時間の片方の改善を実現するために，もう一方を大幅に犠牲にする．

ところが利用者が分散ストリーム処理システムを構築する際には，ネットワークの利用可能なバンド幅に上限が生じる場合やシステムの停止時間の上限が存在するなど，バンド幅オーバヘッドやリカバリ時間に対して制約が付くことは少なくない．それゆえ両指標を柔軟に設定することが求められる．しかし従来方式は，そのような柔軟な設定を許さない為，実際的な利用者の要求を満足しない．

そこで本稿では，Active Standby 方式と Upstream Backup 方式を一般化した高信頼化方式 Lazy Standby を提案する．Active Standby 方式と Upstream Backup 方式が概念的に同じ動作をすることを見出し，上流側のストリーム処理システムとセカンダリとの間の通信にバッチ処理を適応させることにより，Active Standby 方式と Upstream Backup 方式におけるアーキテクチャの違いを解消した．これにより，提案方式では利用者の要求に合わせてバンド幅オーバヘッドとリカバリ時間を柔軟に設定可能にする．また，通信にバッチ処理を利用することで，圧縮処理を行うことが可能になり，送信するデータ量の削減を実現する．本稿では，分散ストリーム処理システムのプロトタイプシステムを用いて提案方式を評価し，提案方式の有効性を示す．

本稿の構成は以下の通りである．まず，2 節で関連研究について述べる．そして 3 節で本研究の提案手法の説明をする．4 節で評価実験について述べる．5 節でまとめと今後の課題について述べる．

表 1　各リカバリタイプにおける処理出力結果の例

リカバリタイプ	障害前			障害後			
障害なし	t_1	t_2	t_3	t_4	t_5	t_6	$\cdots$
Precise	t_1	t_2	t_3	t_4	t_5	t_6	$\cdots$
Gap	t_1	t_2	t_3		t_5	t_6	$\cdots$
Rollback	t_1	t_2	t_3	t_2	t_3	t_4	$\cdots$

2. 関 連 研 究

本節では，これまでに研究された分散ストリームシステムにおける高信頼化手法の研究について述べる．

2.1 リカバリタイプの分類

分散ストリーム処理システムにおける高信頼化では，手法によって障害回復時におけるデータの欠損やデータの重複が生じる場合がある．Hwang らの研究 [6] では，そのような場合を次の 3 種類に分類している．表 1 は，各リカバリタイプで生成される出力結果の例である．この例では，障害前と後に出力するデータを比較することで，リカバリタイプによる動作の違いを示している．障害が起きていない場合では，t_1,t_2,$\cdots$,t_6,$\cdots$ とデータが順に出力される．

- **Precise Recovery**

Precise Recovery は，障害が発生しない場合に出力されるデータと同一のデータ出力を保証する，最も強力なリカバリタイプである．表 1 では，障害なしが t_1,t_2,$\cdots$,t_6,$\cdots$ とデータを出力するのに対し，Precise Recovery も t_1,t_2,$\cdots$,t_6,$\cdots$ と出力する．

- **Gap Recovery**

Gap Recovery は，障害後に出力するデータに欠損の生じ得るリカバリタイプである．表 1 では，障害後に t_4 を出力せず，t_5,t_6 と出力し，データが欠損している．Gap Recovery は，データの欠損が生じるため，データを継続的に監視することを目的とするストリーム処理を行う場合，重要なデータを見落とす可能性があるため相応しくない．

- **Rollback Recovery**

Rollback Recovery は，障害後に出力されるデータに重複が生じるリカバリタイプである．表 1 では，障害前に t_1,t_2,t_3 というデータを出力するが，障害後にも t_2,t_3 を重複して出力する．

これらのリカバリタイプのうち，Rollback Recovery は次

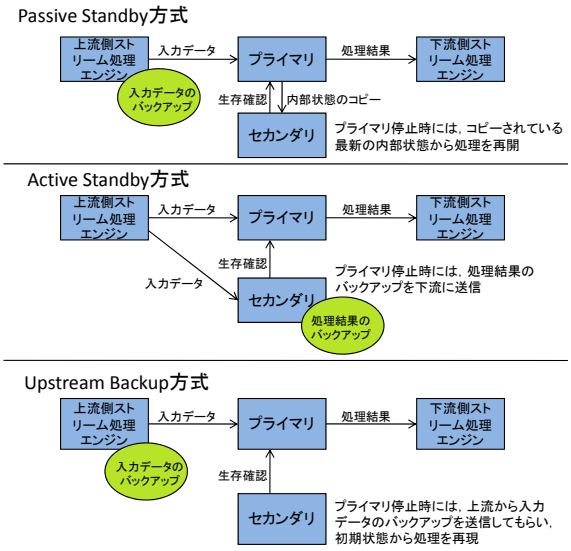

図 1　既存の高信頼化方式

節で説明する 3 つの高信頼化方式で実現が可能である．また，Precise Recovery は，Rollback Recovery に対してデータ重複削除処理を適応させることで実現が可能 [6], [10] である．しかし，Precise Recovery では，重複削除・整合性確認などの処理によって Rollback Recovery よりもバンド幅オーバヘッド・リカバリ時間が大きくなる．

2.2　高信頼化手法の分類

既存の高信頼化手法は，分散ストリーム処理システムを構成する各マシン上のストリーム処理システムに対してバックアップ用のストリーム処理システムを別マシン上に用意し，マシンが停止した場合にはバックアップ用のマシンが故障を検知して問合せ処理を引き継ぐというものである．Hwang らの研究 [6] では，高信頼化手法を以下のように分類している．ここでは，メインで問合せ処理を行うストリーム処理エンジンを**プライマリ**，プライマリの停止時に問合せ処理を引き継ぐストリーム処理エンジンを**セカンダリ**と呼ぶ．また，あるプライマリに対して，プライマリにデータを送信するストリーム処理システムを**上流側ストリーム処理システム**，プライマリから処理結果を送信されるストリーム処理システムを**下流側ストリーム処理システム**と呼ぶ．(図 1)

2.2.1　Passive Standby 方式

Passive Standby 方式 [6], [8] は，プライマリがメインで問合せ処理を行いながら，定期的にプライマリの内部状態をセカンダリへコピーする方式である．プライマリが生存している間は，問合せ処理は行わず，プライマリの生存確認を定期的に行う．プライマリが停止した場合，セカンダリは最新のコピーを用いて処理を再開する．このとき，コピー後にプライマリが処理したデータはセカンダリのコピーに反映されていない為，そのまま再開しては処理データが失われる．この損失を防ぐ為，セカンダリはそれらのデータを処理再開時に上流のストリーム処理システムから再送させる．

2.2.2　Active Standby 方式

Active Standby 方式 [4], [6], [7] では，上流のストリーム処理システムからプライマリが受信するデータを全てセカンダリも受け取り，セカンダリもプライマリに並列して問合せ処理を行う．プライマリが生存している場合には，セカンダリは処理結果を出力しない．プライマリが停止した場合には，セカンダリが自身の出力を下流側のストリーム処理システムに接続し処理結果の配信を行うことにより処理を引き継ぐ．

2.2.3　Upstream Backup 方式

Upstream Backup 方式 [6] は，プライマリが生存している場合には，セカンダリは待機しながらプライマリの生存確認のみ行い，内部状態のコピーやプライマリと並列した問合せ処理は行わない．その代りに，プライマリの上流側のストリーム処理システムは，プライマリへの入力データをバックアップしておく．プライマリが停止した場合は，セカンダリはデータを持たない初期状態から問合せ処理を開始し，上流のストリーム処理システムにバックアップされているすべてのデータを再送してもらい，処理することでプライマリの状態を復旧する．

2.3　3 方式の特性比較と問題

2.3.1　特 性 比 較

Hwang らの研究では，これら 3 方式の特性比較のために，バンド幅オーバヘッドとリカバリ時間という 2 つの指標を用いている [6]．この 2 つの指標は，(バンド幅オーバヘッド)=(バックアップデータ転送に費やしたバンド幅)/(全データ転送に費やしたバンド幅),(リカバリ時間)=(障害を検知から復旧までに費やした時間) と定義され，互いにトレードオフの関係となっている．

これらを用いて上記の 3 方式を比較する．Active Standby 方式ではプライマリへのデータ送信時は必ずセカンダリへバックアップデータを送信することからバンド幅のオーバヘッドは極端に大きくなる．しかし，セカンダリが常にプライマリと同じ状態で待機できるため，リカバリ時間が極めて小さくなる．Upstream Backup 方式では，通常時はバックアップデータの送信を行わず，障害発生時のみデータの送信を行うため，バンド幅オーバヘッドを極端に小さくする．しかし，セカンダリは初期状態から問合せ処理を開始するため，リカバリ時間が非常に大きくなる．Passive Standby 方式は，定期的にプライマリの内部状態をセカンダリへとコピーするため，障害が発生した場合，最新の内部状態から問合せ処理を開始する．そのため，リカバリ時間は Upstream Backup 方式よりも小さくなる．しかし，頻繁に内部状態のコピーを行うため，バンド幅オーバヘッドは Active Standby 方式とほぼ同様の値となる．

2.3.2　問 題 提 起

これらの既存方式は，バンド幅オーバヘッド，リカバリ時間のどちらか一方を犠牲にしなければ，高信頼化を保証することはできない方式である．しかし，実際に利用者が分散環境を構築する際には，使用するマシンの性能やネットワークの帯域・遅延により，バンド幅オーバヘッドやリカバリ時間に制約がつくことは少なくない．そのため，既存方式では利用者の要求に合わせて柔軟な設定をしながら高信頼化を保証することは困難である．

3. 提案手法 Lazy Standby 方式

本節では，我々が提案する，高信頼化手法 Lazy Standby 方式について説明する．

3.1 従来方式の統一化

Active Standby 方式と Upstream Backup 方式に着目する．この両方式は，バックアップデータの蓄積場所に関するアーキテクチャの違いから正反対の特性を持つが，本質的にはバックアップデータをセカンダリで再処理することでプライマリの処理結果を再現するという同じ概念を持った高信頼化方式である．つまり，バンド幅オーバヘッドとリカバリ時間のトレードオフ関係はバックアップデータの所在により決まると考えられる．ゆえに，バックアップデータの所在を制御し，両方式を統一化することができれば，バンド幅オーバヘッドとリカバリ時間の柔軟な調整が可能になると考えられる．

そこで我々は，上流側のストリーム処理システムとセカンダリとの間に通信処理を設けることにより，Active Standby 方式，Upstream Backup 方式を統一化した手法を提案する．提案手法では，効率化のため通信はバッチ処理される．利用者の要求に合わせてバンド幅オーバヘッドとリカバリ時間の調整が可能な高信頼化手法を実現する．本節で提案する分散ストリーム環境の高信頼化手法を Lazy Standby 方式と呼ぶ．

3.2 システムモデル

Lazy Standby 方式では，既存の高信頼化手法と同様に，分散環境を構築する各マシンに対してバックアップ用のマシンを用意する．図 2 で，ノード N_u,N,N_d という 3 つのノードに分散したシステムモデルの例を表す．ノード N に対して，上流側にあるノードを N_u，下流側にあるノードを N_d とし，ストリームデータの流れを赤い矢印で表す．特にここでは，N_u から N に送信されるストリームデータの流れを I_1,I_2 とし，N から N_d を O としている．本稿では，従来研究 [6] と同様に，通信によるデータ順序の入れ替わりや，データの欠落が生じないことを前提とする．

本システムモデルでは，ノード N_u,N,N_d をプライマリとみなして，ノード N_u',N',N_d' をセカンダリとして用意する．セカンダリは定期的にプライマリの生存確認を行い，プライマリの停止を検知した場合セカンダリが処理の引き継ぎを行う（図 2 では N が故障した場合，I_1,I_2,O をそれぞれ I_1',I_2',O' に切り替える.）本稿では従来研究と同様に，プライマリのみが停止故障する single node failure を想定する．そのため，プライマリとセカンダリが停止故障した場合にはシステムが停止故障する．ただし，複数のプライマリが同時に停止故障した場合には，本稿で述べるリカバリ処理によりシステムは停止故障を免れる．

ノード上のストリーム処理システムの構成要素を，インプットキュー部，オペレータ部，アウトプットキュー部の 3 つに抽象化して扱う．インプットキュー部は，情報源や上流側のストリーム処理システムから配信されるデータを受け取りタプルに変換してキューに蓄える．キューに蓄えられたデータは随時，キューの先頭からオペレータ部へと出力される．また，インプットキュー部は定期的に，上流側ノードのアウトプットキュー部に対して Level-0-Ack を返す．Level-0-Ack は，インプットキュー部が最後のどのデータを受け取ったかという情報を示す．オペレータ部は Window 演算やリレーショナル演算による演算の組み合わせによって構成され，処理システムに相当する役割を持つ．インプットキュー部からオペレータ部へと入力されたデータは連続的問合せ [11] によって処理され，アウトプットキュー部へと出力される．アウトプットキュー部は，オペレータ部から受け取った処理結果を下流側のストリーム処理システムに送信し，バックアップをキューに蓄える．このバックアップデータは下流側のセカンダリから要求があった時のみ，キューの先頭から送信される．アウトプットキュー部は下流側ノードのインプットキュー部から Level-0-Ack を受け取り，Level-1-Ack を上流のアウトプットキューに対して送信する．Level-1-Ack は，Level-0-Ack によって示されるデータを生成する入力データの情報を示す．

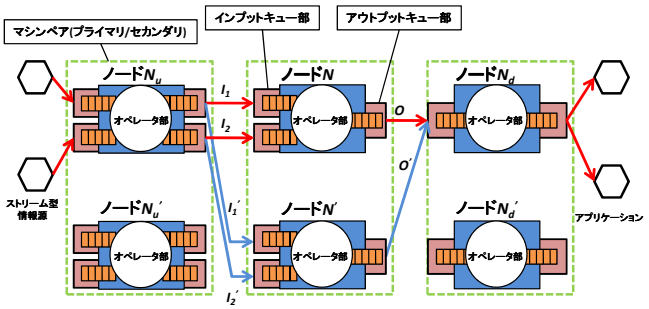

図 2　3 ノードに分散したシステムモデルの例

3.3 Lazy Standby 方式の概要

前節で述べたシステムモデルにおいて，Active Standby 方式はアウトプットキュー部にデータを蓄えることなく下流側のセカンダリへデータを送信し，Upstream Backup 方式ではアウトプットキュー部にデータを理論的には無限に蓄えることで実現される．これに対して，本手法では上流側ノードのアウトプットキュー部とセカンダリのインプットキュー部の間の通信にバッチ処理を用いて通信量を制御することで，両手法を統一化して扱う．通信量 (バッチサイズ) を変化させることで両ノード間に存在するバックアップデータの所在の制御を可能にし，バンド幅オーバヘッドとリカバリ時間の柔軟な調整を実現する．加えて，データをバッチで扱うことによりデータ圧縮が可能になるため，セカンダリへ送信するデータ量を削減し，従来手法よりもバンド幅オーバヘッドを小さくできる．また，本手法では，上流側のノードとセカンダリノードの両方でバックアップデータを保存する．それぞれに蓄積された不要なバックアップデータの削除のために，上流側のノードでは Level-1-Ack，セカンダリノードでは Level-0-Ack を用いる．

本手法のアルゴリズムを 3.4 節，3.5 節で述べる．3.4 節では Lazy Standby 方式におけるバックアップデータの保存アルゴリズムを説明する．3.5 節では，保存されたバックアップデータの削除アルゴリズムを説明する．また，アルゴリズムの説明には図 2 を用いる．ここでは，ノード N が停止する場合を考える．

3.4 バックアップデータの保存

バックアップデータの保存を行うアルゴリズムを以下に示す.

Algorithm1:バックアップデータの保存 (N_u のアウトプットキュー部)

```
1: B：バッチ
2: B_size：バッチサイズ
3: B'：圧縮されたバッチ
4: OQ_u = {t_0, t_1, t_2, ···}：N_u のアウトプットキュー
5: t_i：OQ_u に i 番目に到着したデータ
6: t_new：OQ_u に到着したデータ
7:
8: while TRUE do
9:     if t_new arrived then：t_new を受信
10:        OQ_u.enqueue(t_new)
11:    endif
12:    if |OQ_u| = B_size then
13:        for i = 1 to B_size do
14:            d = OQ_u.dequeue
15:            B ⇐ d：取得した要素をバッチに挿入
16:        end for
17:        B' = compress(B)：B を圧縮
18:        send(B')：セカンダリノードへ B' を送信
19:    endif
20: end while
```

Algorithm2:バックアップデータの保存 (N' のインプットキュー部)

```
1: B'：圧縮されたバッチ
2: B：バッチ
3: IQ'：N' のインプットキュー
4:
5: while TRUE do
6:     if B' arrived then
7:         B = decompress(B')：B' を受信後 B' を解凍
8:         for i = 1 to |B| do
9:             t = B.get(i)：バッチに i 番目に挿入されたデータを取得
10:                IQ'.enqueue(t)
11:            end for
12:    end if
13: end while
```

N_u のアウトプットキュー OQ_u に対してバッチ処理を施すために，バッチサイズ B_{size} を与える．システム開始後，OQ_u はオペレータ部から処理結果を取得するたびに，結果を OQ_u に追加する．OQ_u のキューサイズが B_{size} よりも大きくなったとき，OQ_u の先頭から B_{size} 個のデータを取り出し，バッチ化する．バッチ化が完了したらバッチ B に圧縮処理を施す．ここで圧縮されたバッチを B' とする．B' が生成されたら，B' を N' へ送信する．N' は B' を受け取ったら解凍処理を施し，B を復元する．そして，B をバッチ化前の個別データ t_i,···,$(i+t_{B_{size}}-1)$ に戻し，N' のインプットキュー IQ' へ順に入力する．そして処理を行い，処理結果を N'の アウトプットキュー OQ' へ保存し待機する．

このアルゴリズムは，N_u の OQ_u に保存されるバックアップデータを B_{size} 個単位で切り出して N' へ送信するという振る舞いをする．つまり，B_{size} を変化させることで，バックアップデータの所在の制御できる．これにより，バンド幅オーバヘッドとリカバリ時間の柔軟な調整を実現できる．このアルゴリズムには次の性質がある．

$$\text{Lazy Standby}=\begin{cases}\text{Active Standby} & (iff\ B_{size}=1)\\ \text{Upstream Backup} & (iff\ B_{size}=\infty)\end{cases}$$

B_{size} を 1 にすると，N_u にバックアップデータを保存することなく，N' へと送信し続け，Active Standby 方式ととして動作する．また，同様に B_{size} を ∞ にすると，N' へバックアップデータを送信することなく，N_u で保存し続け，Upstream Backup 方式として動作する．ゆえに提案手法は，Active Standby 方式，Upstream Backup 方式の一般化を実現し，統一的に扱うことができる．

3.5 バックアップデータの削除

本手法では，バックアップデータを OQ'，OQ_u に保存する．このバックアップデータは，バックアップをしている元のデータが確実に N_d へ送信されたことがわかれば，不要となるデータである．そこで，このような不要になったバックアップデータを削除するアルゴリズムを示す.

3.5.1 N' におけるバックアップデータの削除

Algorithm3:N' におけるバックアップデータの削除

```
1: OQ'：N' のアウトプットキュー
2: Level-0-Ack(t_n)：N_d から配信される t_n に関する
   Level-0-Ack
3:
4: while TRUE do
5:     if N' gets Level-0-Ack(t_n) then：N' が Level-0-Ack
       t_n を受信
6:         for i = 1 to |OQ'| do
7:             if OQ'.firstElement older than t_n then
8:                 OQ'.dequeue：OQ' の先頭要素が t_n
                   よりも古いものならキューから削除
9:             else
10:                break
11:            endif
12:        end for
13:    end if
14: end while
```

N' の OQ' では，N_d から定期的に配信される Level-0-Ack を受け取る．Level-0-Ack は，N_d が最後に受信したデータを通知する．そして，OQ' に保存されているバックアップデータと Level-0-Ack の示すデータの比較を行い，Level-0-Ack によって得られたデータよりも古いデータをキューの先頭から削除する．OQ' に蓄積されるバックアップデータの削除を可能にする

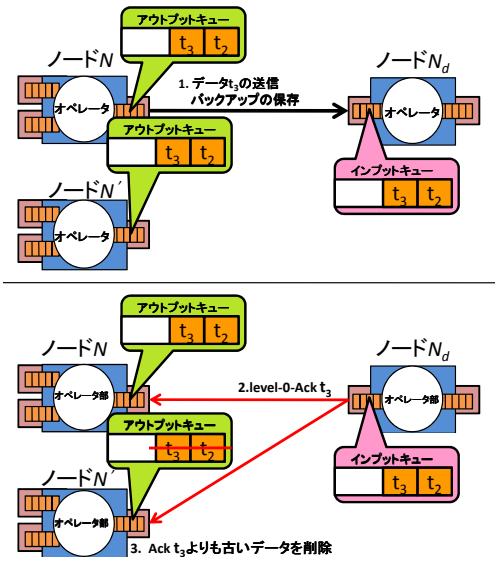

図 3　N' におけるバックアップデータの削除

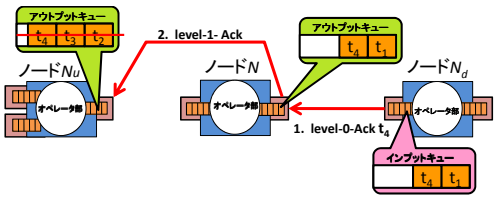

図 4　N_u におけるバックアップデータの削除

ことで，障害発生時に冗長なデータを下流側へ配信する可能性を低くできる．

図 3 で N' におけるバックアップデータの削除の例を示す．まず，N が N_d へデータ t_3 を送信する．そして，それに応じて，データ t_3 の Level-0-Ack が N，N' のアウトプットキューに送信される．この Level-0-Ack によって，N' のアウトプットキューに保存されている t_3 よりも古いバックアップデータは不要になる．そこで，t_3 より古いデータを削除する．

3.5.2　N_u におけるバックアップデータの削除

Algorithm4: N_u におけるバックアップデータの削除

1: OQ_u：N_u のアウトプットキュー
2: t_n：N_u が最後に受信したデータ
3: Level-1-Ack(t_n)：N から配信される t_n に関する Level-1-Ack
4:
5: while TRUE do
6:　　if N_u *gets* Level-1-Ack(t_n) then：N_u が Level-1-Ackt_n を受信
7:　　for $i = 1$ to $|OQ_u|$ do
8:　　if $OQ_u.firstElement$ older than t_n then
9:　　$OQ_u.dequeue$：OQ_u の先頭要素が t_n よりも古いものならキューから削除
10:　　else
11:　　break
12:　　end if
13:　　end for
14:　　end if
15: end while

N_u にあるバックアップデータの削除を行うためには，バックアップデータのうち，どのデータが既に N_d に到着しているかを取得する必要がある．そこで，N は N_d から定期的に配信される Level-0-Ack を受け取る毎に，Level-1-Ack を生成し，それを N_u へ送信する．Level-1-Ack とは，Level-0-Ack の示すデータを生成するために必要な入力データを通知する．N_u は Level-1-Ack を受け取ることにより，不要なバックアップデータの判別をする．これにより，Level-0-Ack によって得られたデータよりも古いデータをキューの先頭から削除する．上流ノードに蓄積されるバックアップデータの削除を可能にすることで，N' に送信するバックアップデータを削減することができる．

図 4 で N_u におけるバックアップデータ削除の例を示す．まず，N が N_d から，データ t_4 の Level-0-Ack を受け取る．N の OQ はそれに応じて，データ t_4 の Level-1-Ack を N_u へと送信する．Level-1-Ack を受け取った N_u の OQ_u は t_4 よりも古いバックアップデータの削除を行う．

3.6　リカバリ処理

バッチ処理を用いたバックアップデータの送信制御を行うことにより，N' は，バッチとして配信されたデータまでは完全に復旧できるが，バッチとして配信されていないデータは，N_u の OQ_u が保存しているため，再送してもらう必要がある．そこで，提案手法では，障害発生後 N' と N_u，N_d の接続を確立した後，OQ' は，キューに保存しているバックアップデータを N_u へ送信し，N' のインプットキュー IQ' は，N_u の OQ_u に保存されているバックアップデータの送信要求を行い，データを受け取り処理を再現する．本手法では，従来手法と同様に，すでに下流側のノード N_d に到着したデータに関して Level-0-Ack を返す前に障害が発生するなどして，バックアップデータの削除が常に最適に行われるとは限らないため，Rollback Recovery を保証する．ただし，障害回復処理を行う前に，下流側ノードが既に受信しているデータの確認・該当バックアップデータの削除を行うことで Precise Recovery も保証できる．

4.　評 価 実 験

本節ではプロトタイプシステムを用いた Lazy Standby 方式の評価実験について述べる．

4.1　実 験 環 境

本研究で行った評価実験の実験環境について述べる．本実験では，選択演算のみが実行可能なプロトタイプシステムを実装し，6 台のマシンを用いて図 5 のような構成の実験環境を構築した．node1 は情報源として疑似ストリームデータを生成し，node2 へ配信する．node2,3,4,5 ではプロトタイプシステムを実行し，高信頼化環境作る．この高信頼化環境では，node3 が障害で停止する場合を想定している．node4 は node3 の停止を確認したのち，障害回復処理を行う．また，node2,3,4,5 の各マシンはそれぞれ図 2 における N_u，N，N'，N_d に該当する．

node1〜node6

CPU	Xeon 2.4GHz
メモリ	2GB
LAN	1000BaseT
OS	Linux(2.6.22)

パラメータ

データレート	100(tuple/s)
生存確認間隔	1(ms)
Level-0-Ack送信間隔	250(ms)
処理時間	10(ms/tuple)

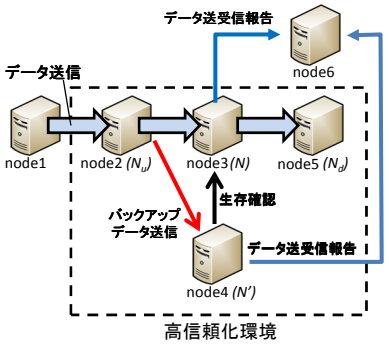

図 5　実 験 環 境

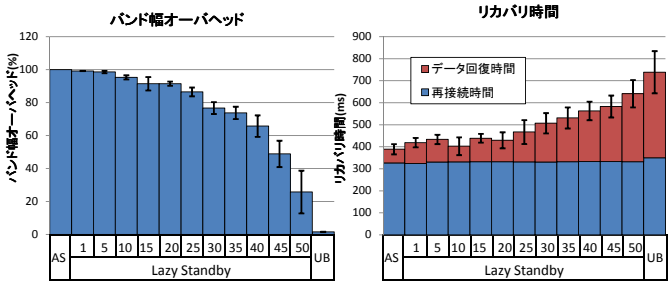

図 6　圧縮なし Lazy Standby 方式の実験結果

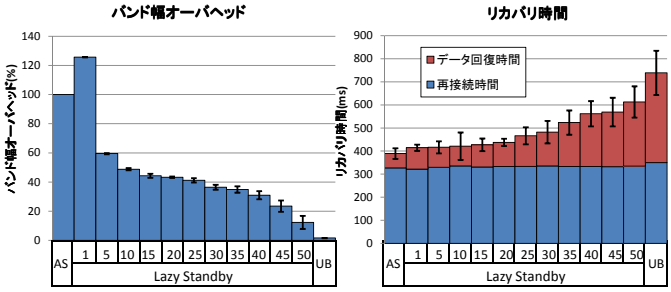

図 7　圧縮あり Lazy Standby 方式の実験結果

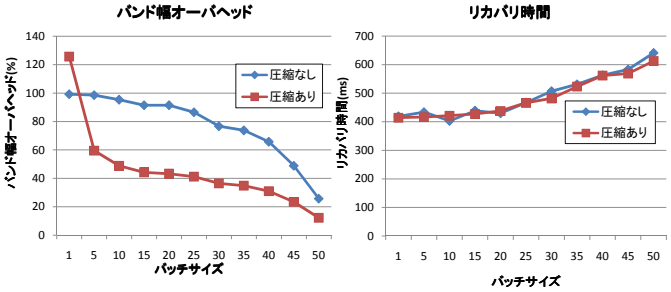

図 8　圧縮なし方式と圧縮あり方式の比較

node6 は node2,4 からデータ送受信報告を受け取り記録する．

4.2　実　　験

本実験では，提案手法 Lazy Standby 方式の B_{size} を変化させ，それぞれの B_{size} におけるバンド幅オーバヘッドとリカバリ時間の測定を行った．バンド幅オーバヘッドでは，プライマリ間の送信に必要なバンド幅に対する，バックアップデータの送信に必要なバンド幅の割合を評価し，リカバリ時間では，node4 が node3 の状態を完全に復元するまでに要する時間を評価する．また，リカバリ時間では，node4 が node2，node5 と接続を確立するために必要になる再接続時間と，バックアップデータの送信や再処理などといったデータ回復時間の 2 つに分けて評価をする．本実験では，ストリームデータ配信開始 30 秒後に node3 を停止させたときのデータを測定する．この操作を各 B_{size} ごとに 30 回試行し，その平均値と標準偏差を実験結果とする．また，本実験では，Active Standby 方式 (AS)，Upstream Backup 方式 (UB) でも同様の実験を行い，Lazy Standby 方式との性能比較を行った．実験に用いた B_{size} は $1, 2, 10, 15, 20, 25, 30, 35, 40, 45, 50$ である．

4.2.1　実験 1：圧縮なし Lazy Standby 方式

まず，圧縮を行わない Lazy Standby 方式についての実験結果を図 6 に示す．

図 6 より，Lazy Standby 方式は B_{size} を大きくするのに伴い，バンド幅オーバヘッドを減少させ，リカバリ時間を増大させていることがわかる．また図 6 では，$B_{size} = 1$ のときに Active Standby 方式とほぼ同様の実験結果を示し，B_{size} を大きくするにつれて，Upstream Backup 方式の実験結果へと近似されていくことがわかる．$B_{size} = 1$ のとき，Lazy Standby 方式では，node2 のアウトプットキューに存在する全てのバックアップデータはバッチ化されて node4 へと送信される．そのため，バンド幅オーバヘッドは増大するが，常に node3 と同じ状態で待機できるため，リカバリ時間が小さくなる．これにより，Active Standby 方式と同じバンド幅オーバヘッドを示していると考えられる．これに対して，B_{size} を大きくすることで，node2 のアウトプットキューに B_{size} 分のデータが蓄積されるのを待たなくてはならない．これにより，B_{size} が大きいほど，node2 のアウトプットキューに蓄積されるバックアップデータが増加し，node4 に保存されるバックアップデータが減少することになる．そのため，バンド幅オーバヘッドは減少する．また，上流に蓄積されたデータはすべて node4 で再処理されるため，リカバリ時間が増大する．ゆえに，B_{size} の増加に伴い提案手法 Lazy Standby 方式は Upstream Backup 方式に近似した実験結果を示すと考えられる．

4.2.2　実験 2：圧縮あり Lazy Standby 方式

次に圧縮を行う Lazy Standby 方式についての実験結果を以下に示す．なお，本実験では圧縮ライブラリ zlib [17] を用いて圧縮を行った．

図 7 より，B_{size} の変化による，バンド幅オーバヘッドとリカバリ時間の変化傾向は，圧縮を用いなかった場合とほぼ同様の結果を示すことがわかる．一方，図 7 の $B_{size} = 1$ のときにおいて，バンド幅オーバヘッドが大幅に大きくなっていることがわかる．この理由は圧縮ライブラリ zlib の使用によるものである．zlib は圧縮を行う際，圧縮するデータの先頭にヘッダを付加する．これにより，圧縮するデータが極めて小さい場合では，圧縮後のデータサイズが，圧縮前のデータサイズよりも大きくなってしまうという現象が起きる．本実験でも，$B_{size} = 1$ のときに配信されるデータサイズが，圧縮前は 33byte であったのに対して，圧縮後に 41byte となっていること確認できた．

図 8 で，圧縮を用いた方式と用いない方式の比較結果を示す．

この結果より，圧縮あり Lazy Standby 方式は $B_{size} = 1$ のときにバンド幅オーバヘッドが極めて大きくなってしまうものの，圧縮なし Lazy Standby 方式に比べて，概ね 45%〜60%程度にバンド幅オーバヘッドを小さくしていることがわかる．これに対して，両方式におけるリカバリ時間の変動は，圧縮なし Lazy Standby 方式，圧縮あり Lazy Standby 方式の両方式ともほぼ同様のリカバリ時間を示すことがわかる．

4.3 実験のまとめ

前節で，提案手法 Lazy Standby 方式はバッチ処理を用いることにより，従来手法 Active Standby 方式，Upstream Backup 方式を統一化し，バンド幅オーバヘッドとリカバリ時間の柔軟な調整を可能にすることを示した．また，本手法はバッチ処理を用いることで，送信データに対して圧縮をすることができる．本手法では圧縮を用いることによりバンド幅オーバヘッドを，圧縮を用いない場合に対して概ね 45%〜60%程の大きさに削減することを示した．

提案手法 Lazy Standby 方式は，バンド幅オーバヘッドとリカバリ時間の関係をほぼ自由に調整することが可能である．リカバリ時間やバンド幅オーバヘッドに制限があるような状況で分散ストリーム処理システムの高信頼化を実現したい場合，従来手法ではバンド幅オーバヘッド・リカバリ時間のどちらか一方を極端に小さくし，他方を極端に大きくすることしかできなかった．これに対して Lazy Standby 方式では，バンド幅オーバヘッド・リカバリ時間を極端に大きくすることなく，分散ストリーム処理システムの高信頼化を実現することが可能である．それゆえに，Lazy Standby 方式は従来手法よりも，現実に則した高信頼化手法である．

また提案手法 Lazy Standby 方式は，前節の実験結果より，圧縮処理を用いることによって大幅にバンド幅オーバヘッドを削減しながら，圧縮処理を用いない場合とほぼ同様のリカバリ時間で回復処理を行うことが分かる．したがって，バンド幅オーバヘッドを小さくしたい場合には，圧縮なし Lazy Standby 方式よりも圧縮あり Lazy Standby 方式の方が効果的である．

5. まとめと今後の課題

本研究では，上流側ノード・セカンダリノード間のバックアップデータ送信に対してバッチ処理を取り入れ，従来の高信頼化手法である Active Standby 方式と Upstream Backup 方式のアーキテクチャを統合する手法，Lazy Standby 方式を提案した．本稿では，バッチサイズを調整することによって，本手法がリカバリ時間とバンド幅オーバヘッドの調節が可能であることを，プロトタイプシステムによる実験で示した．

また，本手法は，バックアップデータの送信に対してバッチ処理を用いることにより，バックアップデータに対して圧縮を行うことを可能にする．本稿では，バックアップデータの送信に対して圧縮処理を用いることによって，バンド幅オーバヘッドを極端に削減することができることをプロトタイプシステムによる実験で示した．

今後の課題は，各ノードにおける CPU コストの評価，Passive Standby 方式との比較実験，より本格的なストリーム処理システムを用いての評価実験が挙げられる．

提案手法 Lazy Standby 方式は圧縮を用いることにより低バンド幅な高信頼化を実現することが可能であるが，圧縮・解凍処理を用いることによって従来手法よりも CPU 使用率が上昇することが考えられる．そのため，各ノードにおける CPU コストの評価をする必要がある．加えて，本稿では提案手法 Lazy Standby 方式が一般化している従来手法 Active Standby 方式と Upstream Backup 方式との比較実験は行っているが，Passive Standby 方式との比較は行っていない．したがって，Lazy Standby 方式と Passive Standby 方式の比較実験を行う必要がある．また，本稿の評価実験では，選択演算のみが実行可能な分散ストリーム処理システムを用いているため，現実で使われる本格的な分散ストリーム処理システムを用いた評価実験をする必要があると考えている．

謝辞 本研究の一部は，科学研究費補助金 (#18200005)，科学技術振興機構 CREST「自律連合型基盤システムの構築」，筑波大学 VBL 研究プロジェクトによる．

文　献

[1] D.J.Abadi et al.”The Design of the Borealis Stream Processing Engine,” Proc. CIDR,pp.277-289,2005.
[2] D.Gyllstrom et al.”SASE: Complex Event Processing over Streams,” Proc. CIDR,2007.
[3] Alan J.Demers et al.”Cayuga: A General Purpose Event Monitoring System,” Proc. CIDR,2007.
[4] S.Chandrasekaran et al.”TelegraphCQ: Continuous Dataflow Processing for an Uncertain World,” Proc. CIDR,2003.
[5] M.A.Shah et al.”High-Available, Fault-Tolerant, Parallel Dataflows,” Proc. ACM SIGMOD,pp.827-838,2004.
[6] J.Hwang et al.”High-Availability Algorithms for Distributed Stream Processing,” Proc. ICDE,pp.779-790,2005.
[7] M.Balazinska et al.”Fault-Tolerance in the Borealis Distributed Stream Processing System,” Proc.ACM SIGMOD,2005.
[8] J.Hwang, et al.”A Cooperative, Self-Configuring High-Availability Solution for Stream Processing,” Proc. ICDE,pp.176-185,2007.
[9] J.Hwang et al.”Fast and Highly-Available Stream Processing over Wide Area Networks,” Proc. ICDE,2008
[10] J.Hwang et al.”High-availability algorithms for distributed stream processing”, Technical Report CS-04-05,Department of Computer Science,Brown University,2004.
[11] A.Arasu et al. ”The CQL Continuous Query Language: Semantic Foundations and Query Execution,” VLDBJournal,vol.15,no.2,2006.
[12] 稲守孝之, 渡辺陽介, 北川博之, 天笠俊之, 川島英之.「広域分散ストリーム処理環境における演算配置最適化手法の評価」,DEWS2008,2007 年.
[13] 渡辺陽介, 北川博之.「分散環境におけるストリーム処理の高信頼化」情報処理学会研究報告 Vol.2006,No.78(2006-DBS-140(II)),pp.261-268,2006 年 7 月
[14] StreamSpinner. http://www.streamspinner.org/
[15] CORAL8,INC. http://www.coral8.com/
[16] ”Building Distributed Applications Complex Event Processing with Coral8,” http://download.microsoft.com/download/c/b/a/cba8b11b-9143-4f99-916a-1dd6516b3192/ComplexEventProcessingCoral8.pdf
[17] zlib.http://www.zlib.net/