ベンチマークレポート

ーデータグリッド Coherence 編ー

平成 22 年 9 月

グリッド協議会　先端金融テクノロジー研究会　ベンチマーク WG

All Rights Reserved, Copyright © グリッド協議会, 2010

《 目 次 》

ーデータグリッド Coherence 編ー 1

1. COHERENCE (ORACLE) 2

1.1 Coherence の機能概要 2

1.2 Coherence の評価結果 3

1.2.1 評価シナリオ0：事前テスト 3

1.2.2 評価シナリオ1：スケーラビリティ 6

1.2.3 評価シナリオ2：スループット 9

1.2.4 評価シナリオ3：マーケットデータフィード 13

1. Coherence (Oracle)

1.1 Coherence の機能概要

Oracle Coherence（以下、Coherence）は、Oracle 社が提供するデータグリッド製品である（2007 年の Tangosol 社買収により取得）。2001 年のリリース以降、多くの金融機関および大規模 EC サイトなどで採用され、計算用データグリッドの構築や注文管理システム、統合データサービス、大量リクエストに対応するためのスケールアウト可能なシステム基盤などとして利用されている。（図 1-1 参照）。

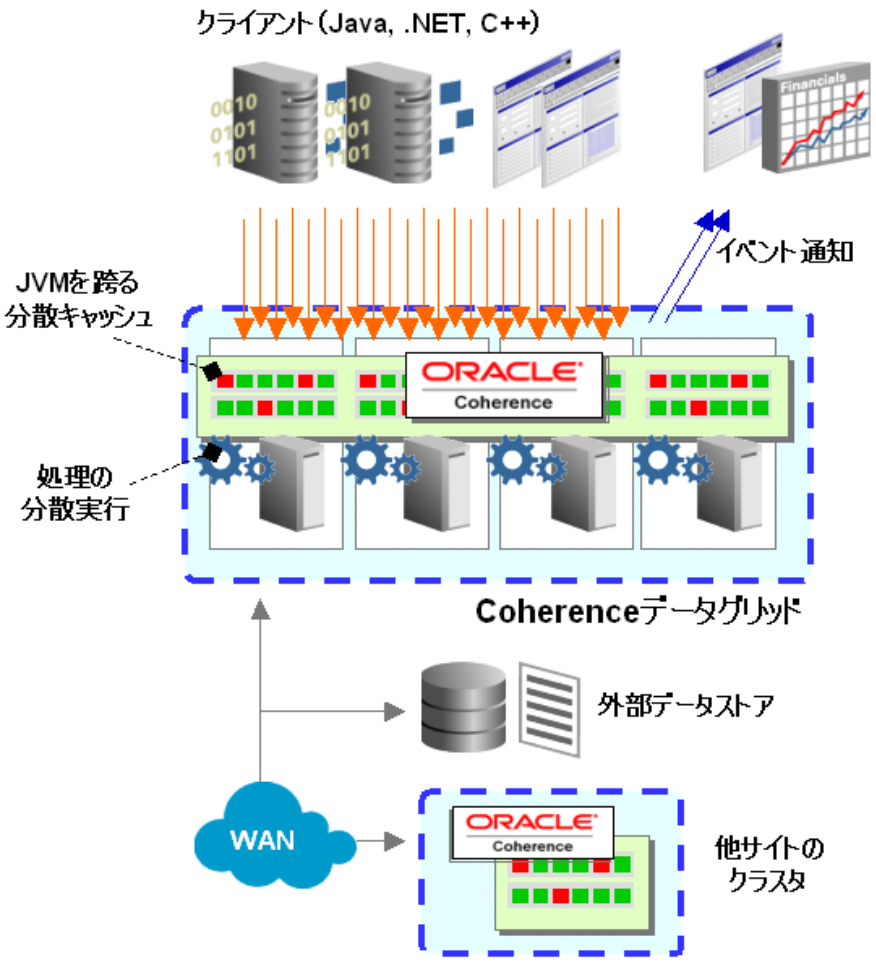

図 1-1 Oracle Coherence の基本アーキテクチャ

Coherence の機能的な特長を以下にまとめる。

(1) 分散キャッシュ

Coherence では Partitioned Cache と呼ばれる機能により、複数のサーバに跨る大規模な分散キャッシュを構成することができる。データのパーティショニング方式に従ってデータが自動的に分散配置されることにより、大量のデータ処理の負荷分散、およびサーバ追加による全体的な性能向上、容量拡張を可能にする。また Partitioned Cache では各サーバのメモリ内でバックアップデータを持合うことにより、サーバおよびプロセス障害でのデータ信頼性を確保している。

Coherence では Partitioned Cache の他にも各サーバで全データを複製する Replicated Cache（主に参照用データの格納に使用）や、クライアント側の一時キャッシュ機能なども提供し、用途に応じて使い分け、もしくは併用することができる。

(2) TCMP（Tangosol Cluster Management Protocol）

Coherence ではデータ送受信のプロトコルとして TCMP と呼ばれる独自プロトコルを使用する。TCMP ではデータ送受信のトランスポートとして TCP よりも軽量な UDP プロトコルを用いることでデータ送受信の性能を高めるとともに、UDP 自体ではサポートされない到達保証や流量制御などの機能を UDP の上位に実装することでデータ送受信の信頼性も確保している。

(3) イベント通知

データ変更のイベントをトリガとして任意の処理を実行するためのイベント通知・受信機能を利用することができる。また、クライアント側で取得されたクエリの結果セットを、データ変更イベントにより自動的に更新するための Continuous Query 機能を提供し、データの更新確認のためのポーリングを使用しない、Push 型のアプリケーションを作成することができる。

(4) In-Place Processing

クライアント側でのデータ処理のためにその都度クライアント側に対象データをコピーするのではなく、データを持つサーバ上でデータ処理を実行することができる。つまり、クライアント-サーバ間でデータ処理に関連するリクエスト数を減らすことができるため、その分のレスポンス向上が期待できる。また、複数件のデータに対する処理の場合は自動的に複数サーバで並列実行され、大量データに対する一括処理および集計操作の負荷分散を行うことができる。

(5) クライアント API

Java 以外にも.NET および C++に対してクライアント API のネイティブライブラリを提供している。

1.2 Coherence の評価結果

ここでは Coherence を使用した場合の各シナリオの評価結果をまとめる。なお、使用したソフトウェアのバージョンは、Coherence 3.5.0 および JDK 6 Update 14 である。

1.2.1 評価シナリオ０：事前テスト

事前テストの一つ目は、データノード数を 8 に、クライアント数は 100 に固定し、オブジェクトサイズを 1KB, 100KB, 1M, 10M と変化させた場合を測定した。測定結果を図 1-2 および図 1-3 に示す。いずれも、1000 回のアクセスにかかった時間をミリ秒単位で図示している。オブジェクトサイズの大きい領域ではデータ転送とアクセス時間がほとんどを占め、実行時間はオブジェクトサイズにほぼ比例する。オブジェクトサイズの小さい領域では、データへのアクセスにかかる処理時間に近付く。例えば、Update の場合は 200 ミリ秒程度であり、実行時間は、

実行時間＝7.2 秒/1MB×データサイズ＋0.2 秒

と近似できる。

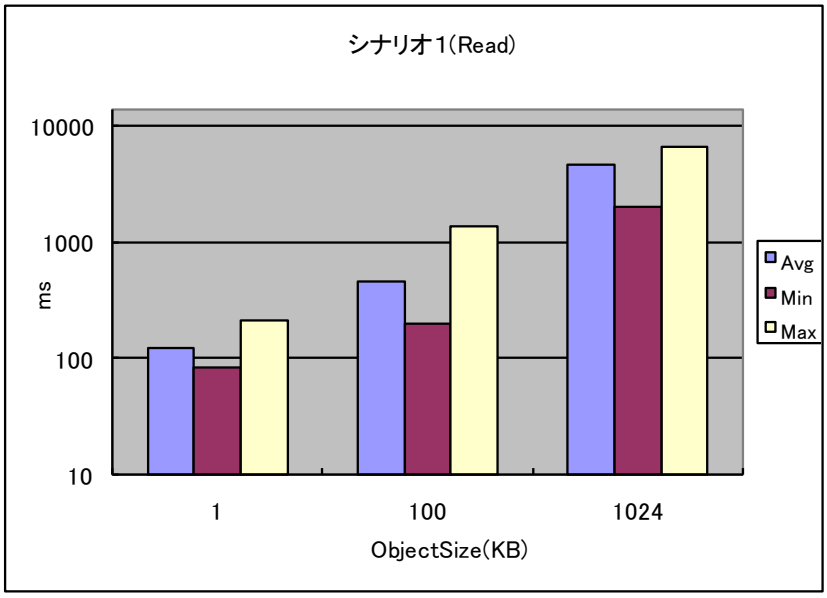

図 1-2 データノード 8、クライアント 100 の場合

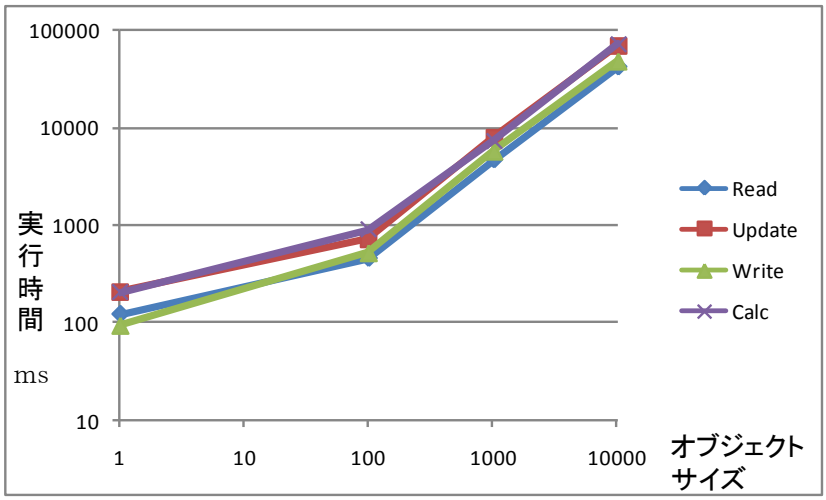

図 1-3 データノード 8、クライアント 100 の場合

事前テストの二つ目は、クライアント数を 100、オブジェクトサイズを 1KB に固定し、データノード数を 1, 4, 8 と変化させた場合を測定した。結果を図 1-4 に示す。1000 回のアクセスにかかった時間をミリ秒単位で図示している。データノード数が 1 つの場合、複数のクライアントからのアクセスが集中・競合し、処理時間が長くなる。データノード数を増やしていくと、CPU ボトルネックである領域では 1 台あたりのデータノードへの負荷が小さくなり、処理時間が短縮される。今回の計測では、データノードが4台以上では、CPU ボトルネックとなっておらず、徐々に無負荷時のレスポンスに近づく。クライアント数をさらに増やすことで、CPU ボトルネックの領域が広がり、データノード4台以上でもデータノードの追加によるレスポンスへの効果は顕著になると推測される。いずれにせよ無負荷時のレスポンスに近づく。無負荷時のレスポンスは、この後の、第三の事前テストの結果から700ミリ秒と推測している。

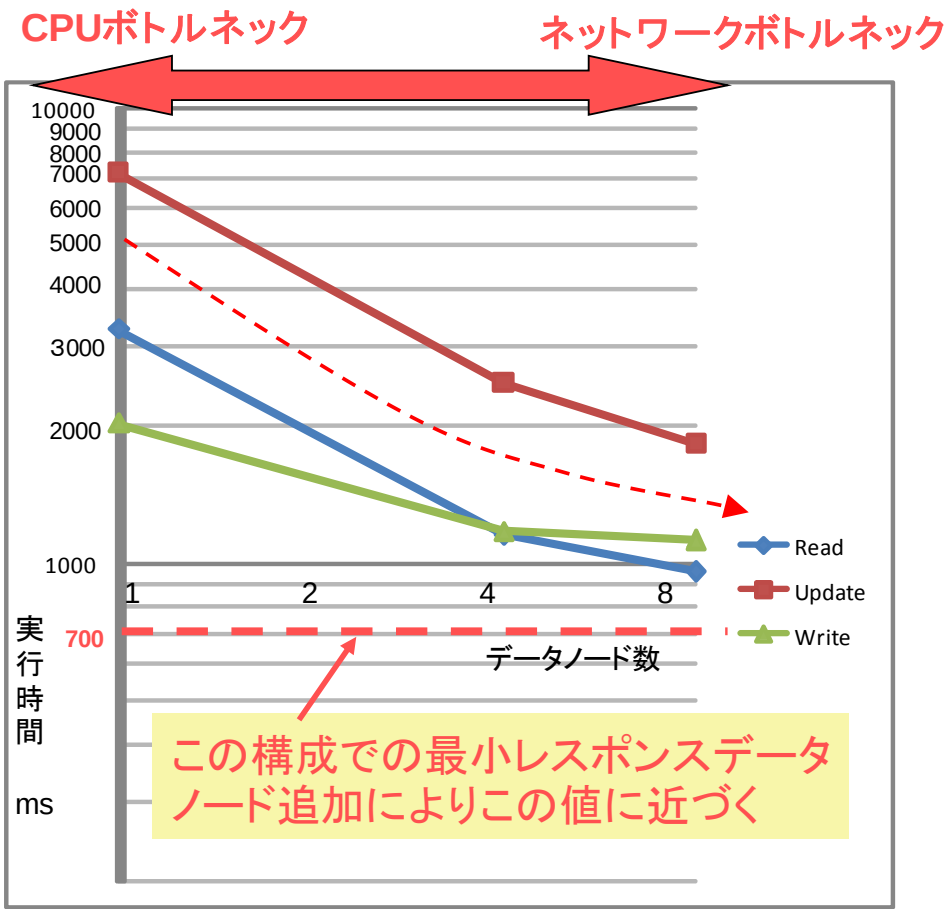

図 1-4 クライアント 100、オブジェクトサイズ 1KB の場合

事前テストの三つ目は、データノード数を 8 に、オブジェクトサイズを 1KB に固定し、クライアント数を 1, 10, 100 と変化させた場合を測定した。結果を図 1-5 に示す。クライアント数 10 程度はアクセス量として負荷が多くないようであり、クライアント数が 100 程度になってアクセスに競合が生じている。無負荷状態でのレスポンスは Read で約 700ms/1000 件と推測できる。この値はリクエスト/レスポンスのネットワーク遅延によるものであり、ボトルネックはネットワークにあるためデータノード数を増やしてもレスポンスは短縮されない。より低レイテンシのネットワークであればレスポンスは短縮される。

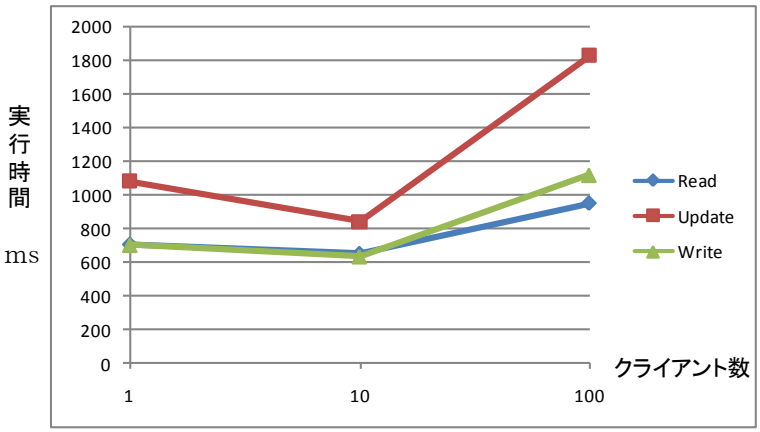

図 1-5 データノード 8、オブジェクトサイズ 1KB の場合

事前テストの結果として、複数データノードに分散することで、1 筐体の物理メモリ以上のデータを保持することが可能であることが分かった。今回の測定では最大で 10GB のデータをデータグリッドに格納した。データアクセスの時間はかなりばらついており、最小と最大で 10 倍近く開く場合もある。これは、起動直後の試行も含んでいることの影響が考えられる。データノード数を 1 から 4 と増やすと、アクセス性能はほぼスケールして向上する。すなわち、ノード数を増やすことによってアクセスの負荷分散が可能であることを示している。タスク数が 100 程度まで増えても、データノードを分散すれば、大きなボトルネックにならずにアクセス可能であることがわかった。

1.2.2 評価シナリオ１：スケーラビリティ

(1) 計測方式

本シナリオではデータ 1,000 件の get/put に要した時間をミリ秒単位で取得した。正確にはクライアント側において 1,000 件の get/put 前後で System.currentTimeMillis()によるタイムスタンプを取得し、その差を処理時間とした。以降、「1 回あたりの処理時間」とは「1000 件の get もしくは put に要した時間」を指すものとする。

最終的な結果のグラフでは、クライアントから得られた処理時間の全サンプルを平均したものを使用する。平均処理時間の算出には各クライアントから 100 回分のサンプルを使用することとし、平均値の算出に使用する 100 回分の実行の前後には、サンプル取得と同数回（つまりここでは 100 回）のウォームアップ、クールダウンの期間を設けた。ただしクライアント数が 1,000 を超えると 100 回の実行に要する時間が長くなることから、クライアントが 1,000 の場合は各クライアント 10～20 回分のサンプルを使用して平均を算出した。

(2) Coherence の構成

キャッシュタイプとして Partitioned Cache（バックアップ数 1）を使用し、シリアライズ方式としては Portable Object Format を使用した。キャッシュのスレッド数等のパラメータはデフォルトのまま使用した。

また、基本的にはサーバ/クライアントともに 1 マシンあたりの JVM 数を 1 とした。ただし検証に使用可能なマシン数に限りがあったことから、クライアント数が 100 および 1000 の場合にはクライアント JVM での実行スレッドを増加させ、合計のスレッド数をクライアント数とした。実際にクライアント側に設定したスレッド数は以下の通りである。

表 1-1 クライアント側 JVM とスレッド数

クライアント数	クライアント側マシン数	マシンあたりの JVM 数	JVM あたりのスレッド数
100	10	1	10
1,000	10	1	100

(3) 測定結果

サーバ数を 2 もしくは 8 で固定し、クライアント数を 1 から 1,000 まで増やした場合の測定結果を以下に示す。

グラフでは平均処理時間とともに、サーバ側の CPU 使用率（vmstat の us,sy の合計値を使用）を折れ線グラフで表示する。データノード数を 2 から 8 へ変えた場合で見てみると、クライアント数が 10 まではサーバに余裕が

あるのでアクセス性能にあまり差がない。しかしクライアント数が 100 を超えるとデータノードでの負荷が高まる。データノード数を 2 から 8 に増やすと 2-3 倍の性能向上が見られ、分散の効果が現れている。

クライアントが 10 程度までだと CPU の稼働率はそれほど高くない。クライアントが 100 を超えると CPU 稼働率が 40%の近辺にへばりついてほとんど変化がない。これはデータアクセスについては CPU の負荷がそれほど大きくなく、ネットワークまたはメモリアクセスがボトルネックになっているためと考えられる。データノードを増やすことによってボトルネックになっていたリソースが増え、全体として処理性能が向上していると考えられる。

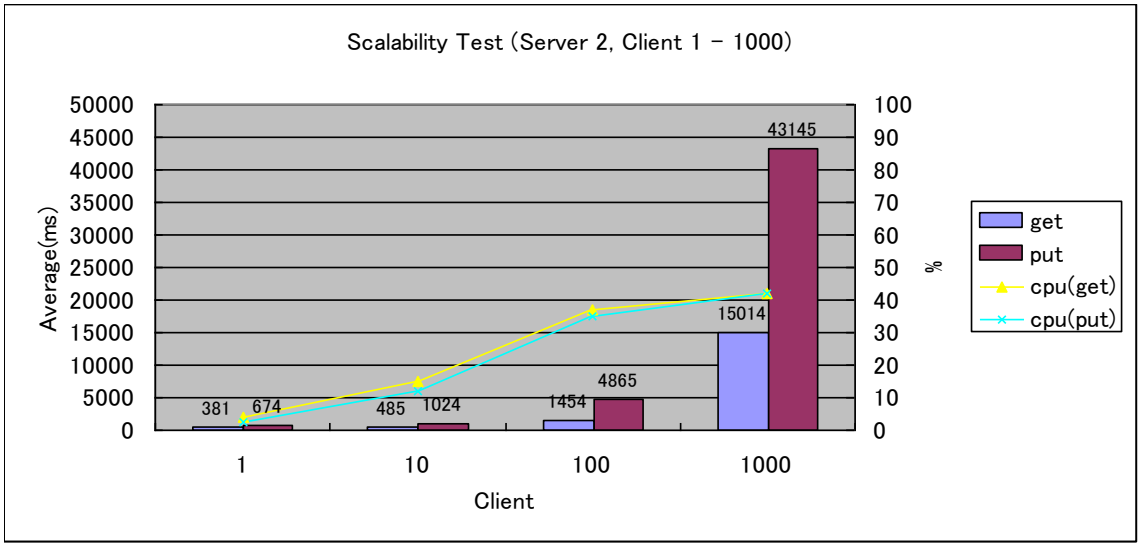

図 1-6　評価シナリオ1（サーバ 2/クライアント 1-1,000）

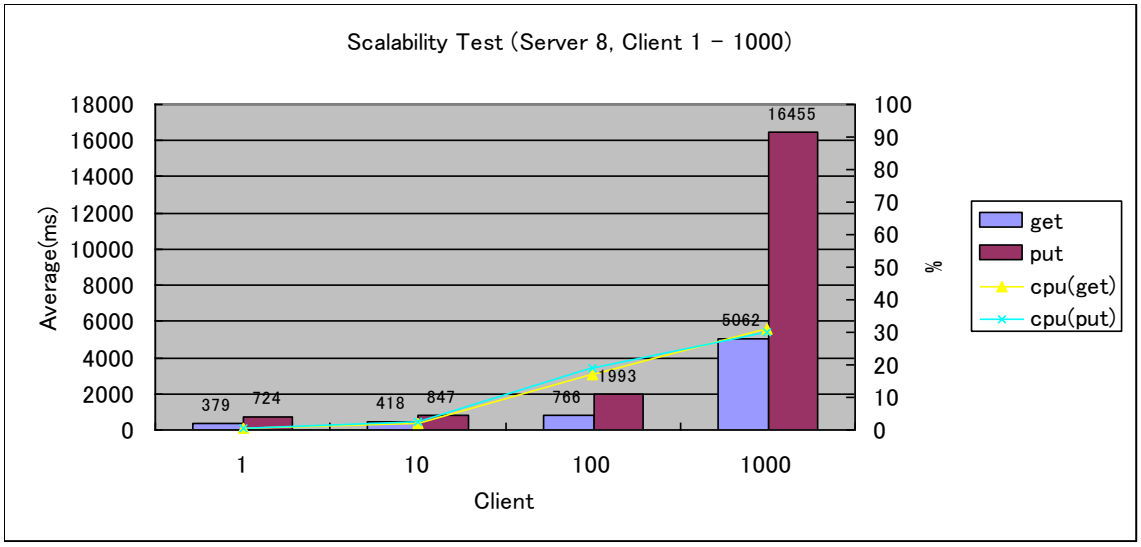

図 1-7　評価シナリオ1（サーバ 8/クライアント 1-1,000）

次に、クライアント数を 100 もしくは 1,000 に固定し、サーバ数を 1 から 8 まで増やした場合の平均処理時間の推移を示す。

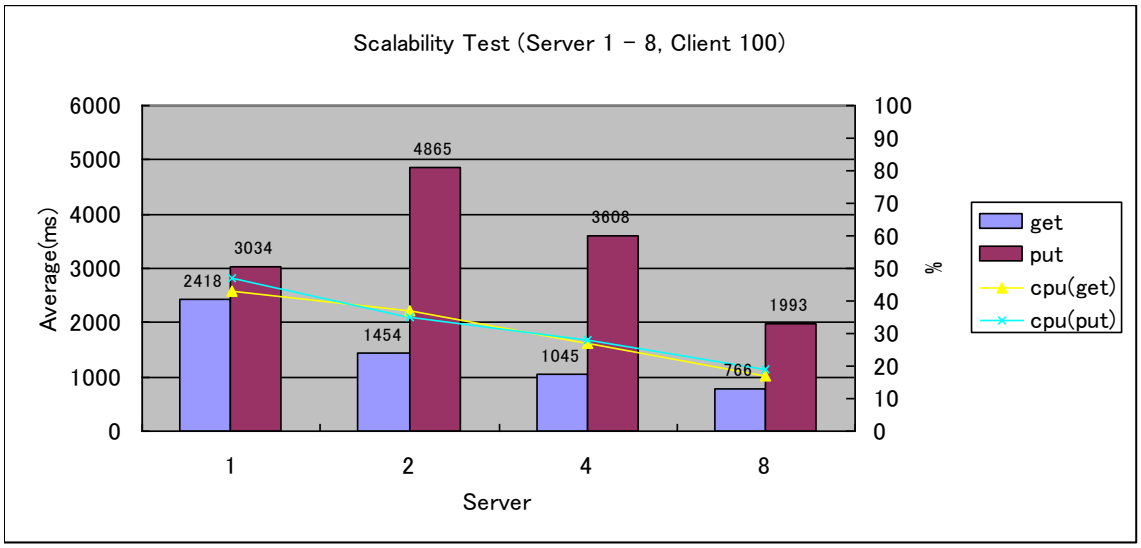

図 1-8　評価シナリオ1（サーバ 1-8/クライアント 100）

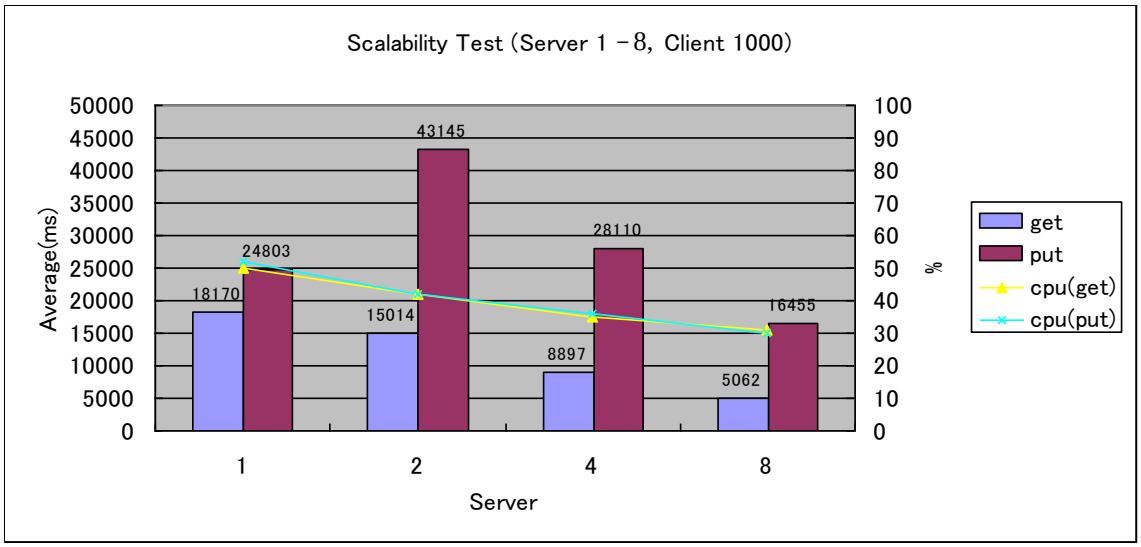

図 1-9　評価シナリオ1（サーバ 1-8/クライアント 1,000）

サーバ数が 1 の場合はバックアップデータが作成されず、put 時にバックアップ更新のオーバヘッドがないために put 性能が傾向と比較して良い結果となっている。サーバ数を増加させた場合の処理時間への影響を考察する上で、サーバ数 1 の put 性能は除外すべきと考えられる。

put および get において、サーバ台数が 2 倍になると処理時間は半分にまではならないものの、短縮される。短縮の度合いは台数が増えるほど鈍くなるが、1.7 倍から 1.3 倍程度の性能向上は得られている。

サーバ台数が1台の場合でも CPU の負荷は 50%程度までであるので、CPU 以外のボトルネックが大きいことを示している。

バックアップがある場合とない場合での put 性能差については別途測定を行った。以下の図はサーバ数 2 でバックアップの有無が put 性能に与える影響を測定したグラフである。

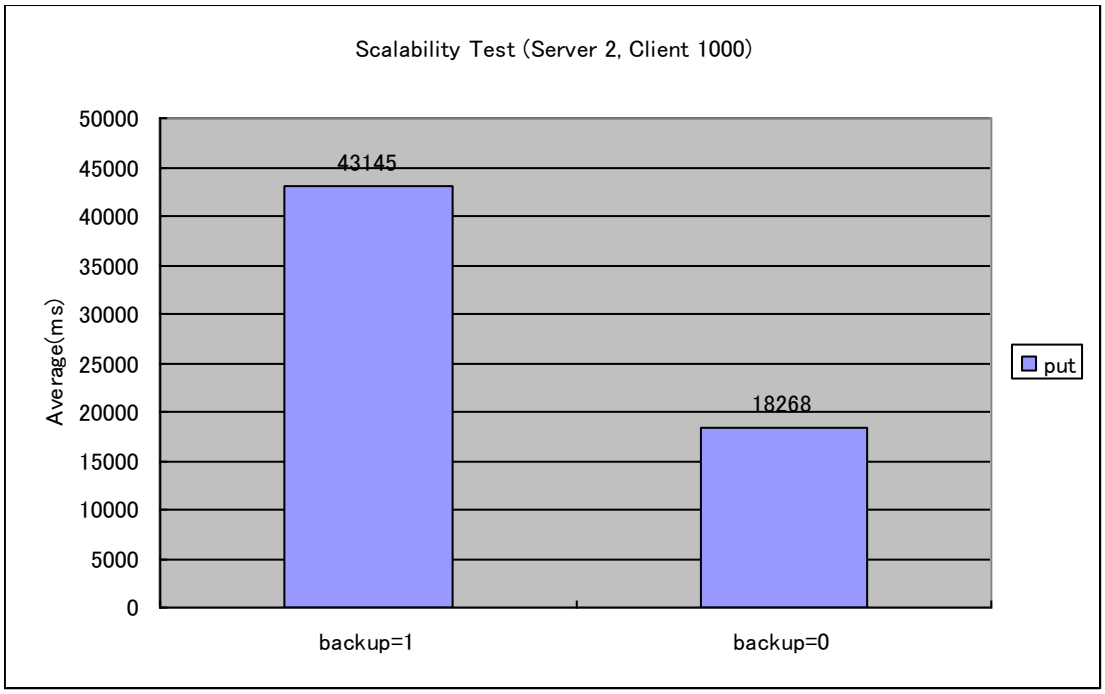

図 **1-10** 評価シナリオ1（バックアップの有無による影響）

データを put した場合にバックアップを作成すると、put 自身にかかる処理の倍以上の時間がかかっている。2回の書き込みに加えて、同期を取るために、処理完了の連絡を待つなどのオーバヘッドがかかるためと推測される。

1.2.3 評価シナリオ２：スループット

(1) 計測方式

評価シナリオ2においても評価シナリオ1と同様の計測を行い、1,000 件の get/put に要した時間とその平均時間を算出した。ただし計測時間短縮の都合上、平均値算出に用いたサンプル数を 2～5 と少なくした。サンプル取得と同数回のウォームアップ、クールダウン期間を設けたのはここでも同様である。

測定結果のグラフでは、平均処理時間から以下の計算式で算出した値をそのシステム全体のスループット（Object/sec）とみなして使用した。

（スループット）=1000÷（1000 件の get/put に要した時間の平均）×（クライアント数）

(2) Coherence の構成

本シナリオではクライアント数が非常に大きく（最大で 6,000）なるため、クライアントの JVM/スレッド数を以下のように構成した。

表 1-2 クライアント側 JVM とスレッド数

クライアント数	クライアント側マシン数	マシンあたりの JVM 数	JVM あたりのスレッド数
100	25	1	4
1,000	25	1	40
2,000	25	1	80
4,000	25	1	160
6,000	25	1	240

サーバ側では、1 マシンあたりの JVM 数がスループットに与える影響についての予備検証を行い、その結果から 1 マシンあたり 3 つの JVM を起動して計測を行った。以下の図では予備検証の結果を示す。

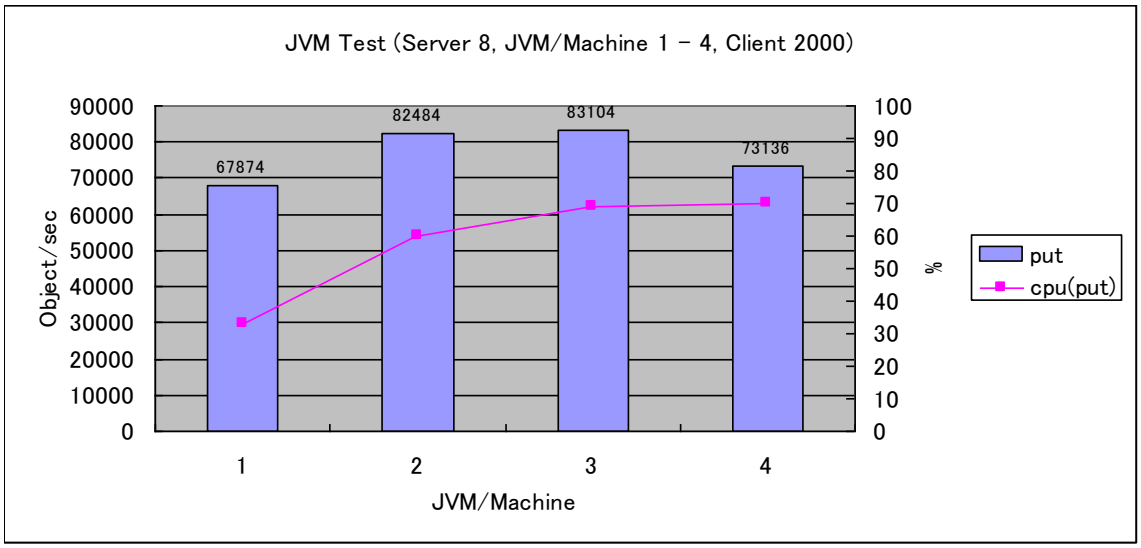

図 1-11 予備検証(マシンあたりの JVM 数とスループット)

予備検証ではサーバ側マシン数を 8、クライアント数を 2,000 に固定し、マシンあたりの JVM 数を 1 から 4 まで増やしてスループットを計測し、スループットが最大となった JVM 数 3 を採用した。最適な JVM 数についてはマシンの持つコア数、およびアプリケーション特性によって変化するため、ここで採用した値が常に最適であるとは限らないことに注意されたい。

(3) 測定結果

まず、サーバ数を固定(2、4、8)し、クライアント数を増加させた際のスループット変化を図で示す。

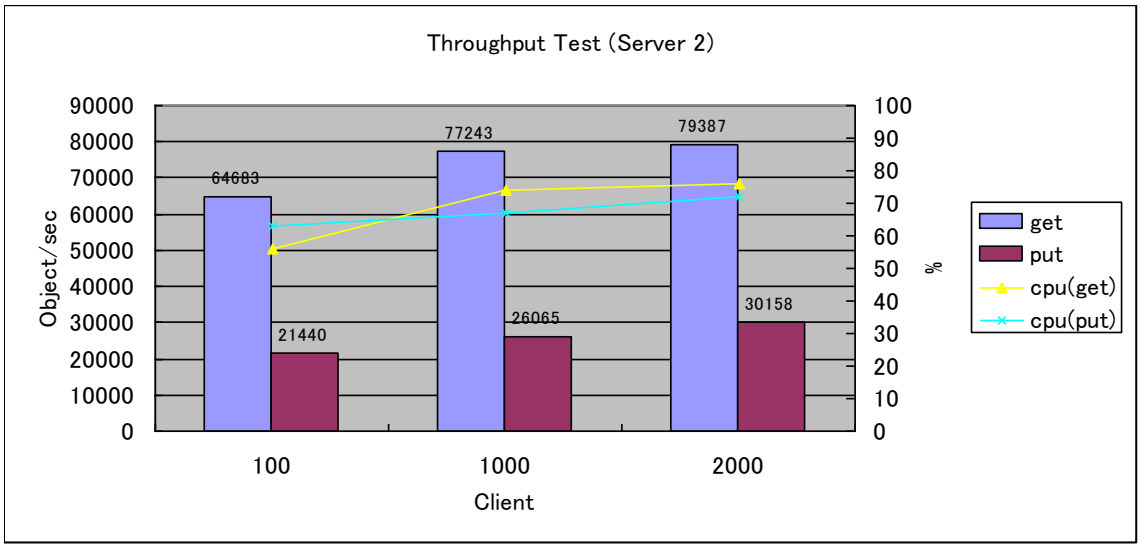

図 1-12　評価シナリオ2（サーバ　2/クライアント 100-2,000）

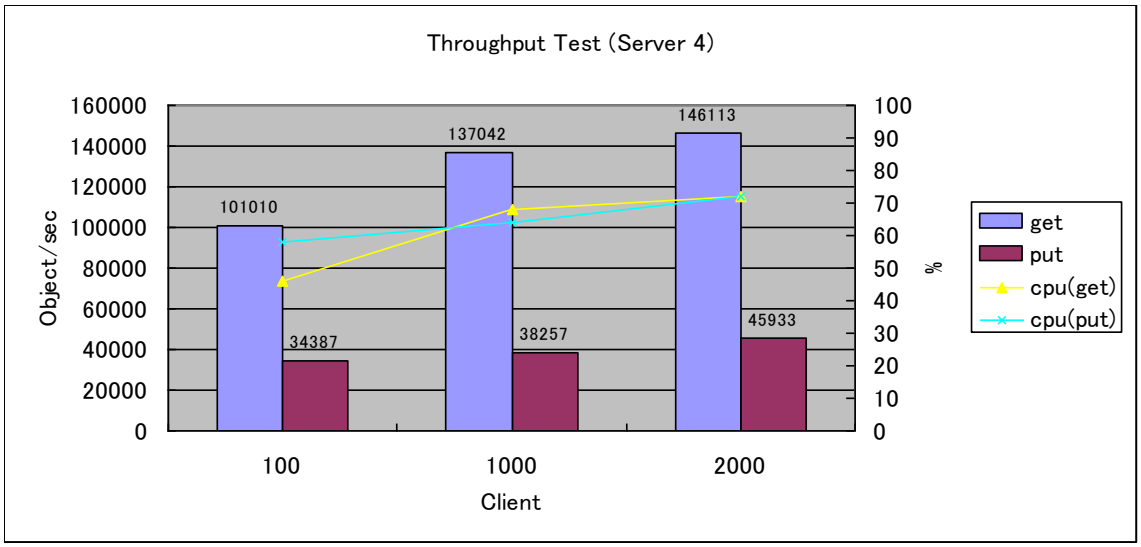

図 1-13　評価シナリオ2（サーバ　4/クライアント 100-2,000）

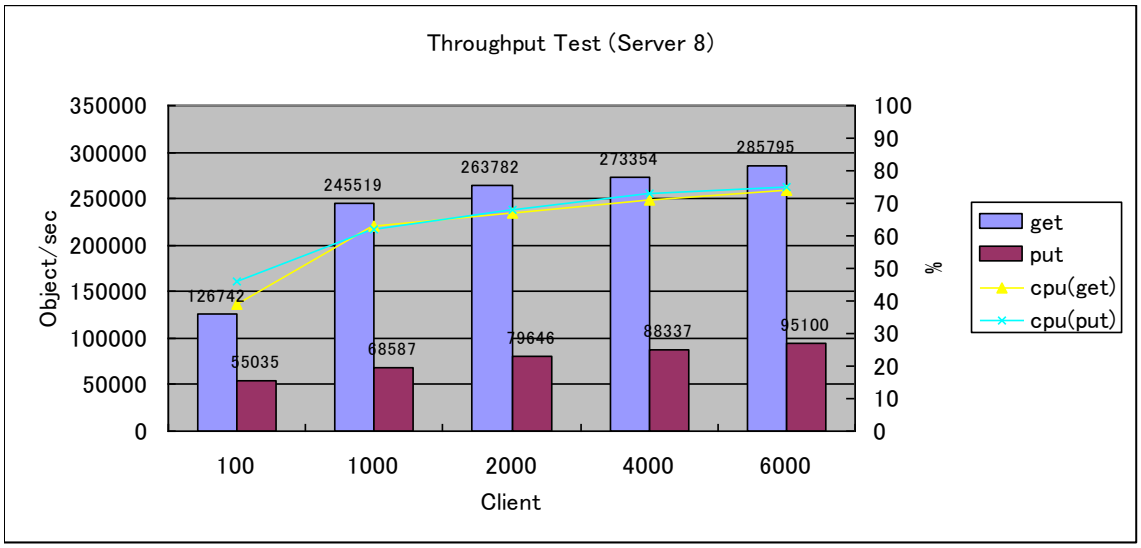

図 1-14　評価シナリオ2（サーバ 8/クライアント 100-6,000）

いずれのサーバ数の場合においても、クライアント数が 1,000 を超えると性能が極端に落ち始める様子が見える。また、ここでは JVM を 3 にしており、シナリオ 1 に比べて CPU 稼働率が高くなっているのが分かる。

それぞれの計測においてクライアント数が最大の時点での性能値をその構成での最大スループットとみなし、サーバ数と最大スループットをプロットした図を以下に示す。

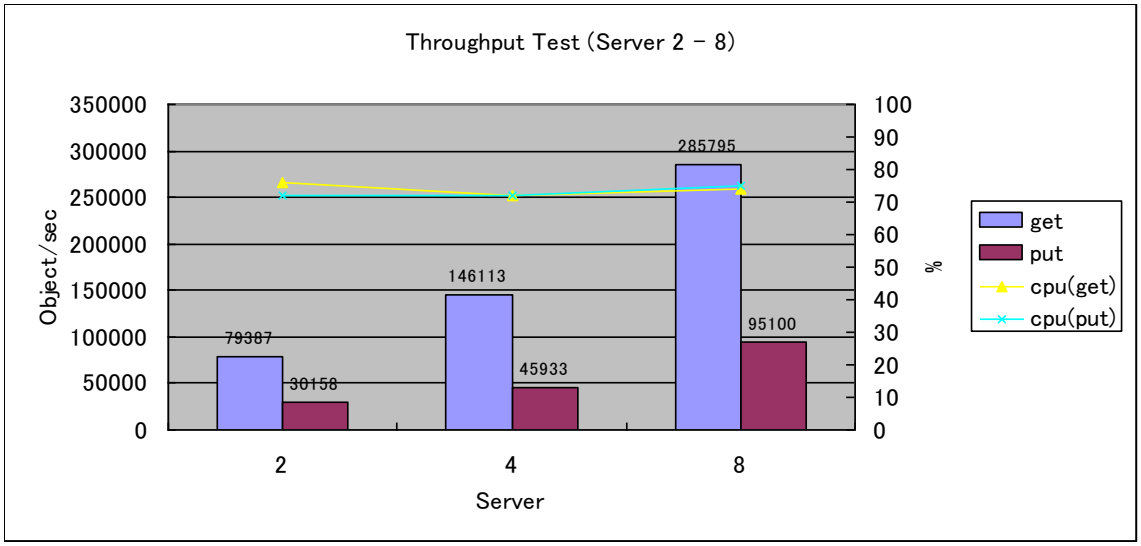

図 1-15　評価シナリオ2スループットの結果（サーバ 2-8）

スループットの性能は非常によくサーバ台数に比例して向上していることが分かる。データを、アクセスする頻度でうまく分散することができれば、サーバ台数を増やすことで効率的に性能アップすることができることを示している。

1.2.4　評価シナリオ３：マーケットデータフィード

(1)　検証用システム/Coherence の構成

評価シナリオ3では以下のような 3 層構成を用いて検証を行った。

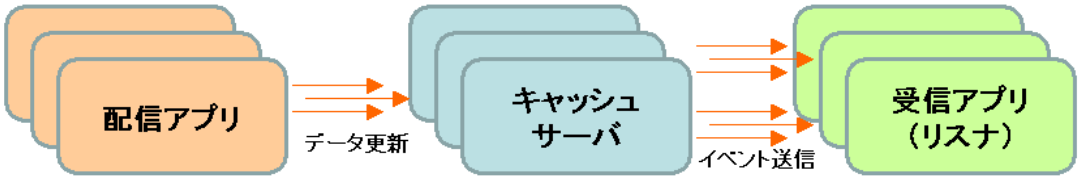

図 1-16　検証用システムの論理構成

銘柄データに相当するデータは Coherence の Partitioned Cache に格納され、配信アプリからデータフィードに相当するデータ更新をキャッシュ上のデータに対して行う。受信アプリは事前に選択された 10 銘柄のイベントを受信するようにリスナを構成し、フィードによるデータ更新をイベントとして受け取る。

キャッシュに格納される銘柄データとしては以下のようなデータを使用した。

表 1-3 使用した銘柄データ

	型	サイズ	備考
Key	String	9 byte	9 文字の銘柄名をそのまま使用
Value 1	Integer	4 byte	株価に相当するデータ
Value 2	byte 配列	124 byte	ダミーのデータ

今回使用したフィードデータには実際の株価は含まれておらず、また検証シナリオにおいて株価の値自体に特に意味はなかったため、株価に相当するデータには初期時に 0、更新時には 1 の値を用いた。ある銘柄データに対して同一の値を用いた複数の更新が発生することになるが、更新前後のデータが同じであったとしてもイベントは発生するため、検証の目的上特に問題はない。

マシン数、Coherence の JVM 数などは以下の値を用いて構成した。

表 1-4 評価シナリオ3 Coherence の構成

	マシン数	マシンあたりの JVM 数	JVM あたりのスレッド数
配信アプリ	8	1	1
キャッシュサーバ	4, 8	1	1
受信アプリ	15	20	1

(2)　検証/計測方式

配信アプリでは QUICK 社より提供いただいた実際のフィードデータを使用した。オリジナルのフィードデータは約 30 分間のデータであるが、単位時間当たりのフィード数を実際よりも大きく設定して検証を行うため、時間を 1/10 もしくは 1/20 に圧縮して配信時刻を算出した。

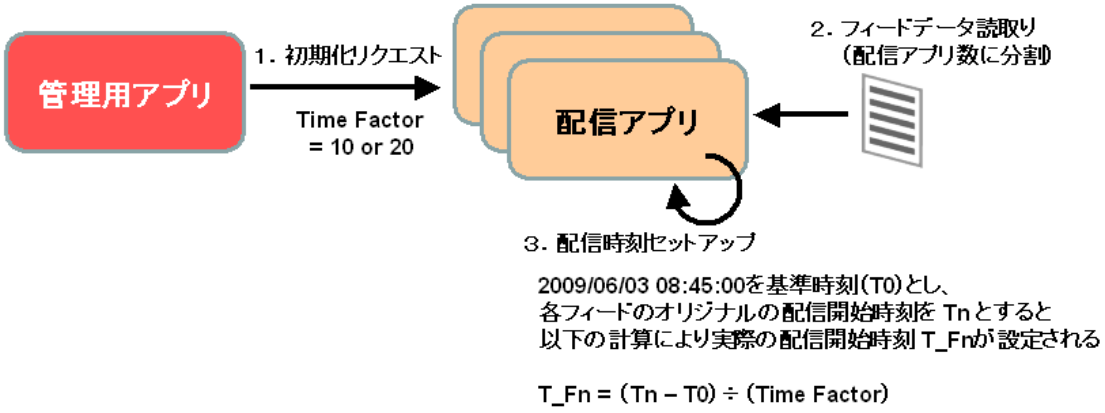

図 1-17　配信時刻の設定

また、計測データとしては配信アプリが各フィードに対して配信を開始した時刻を取得した。時刻データ取得のフローを以下の図に示す。

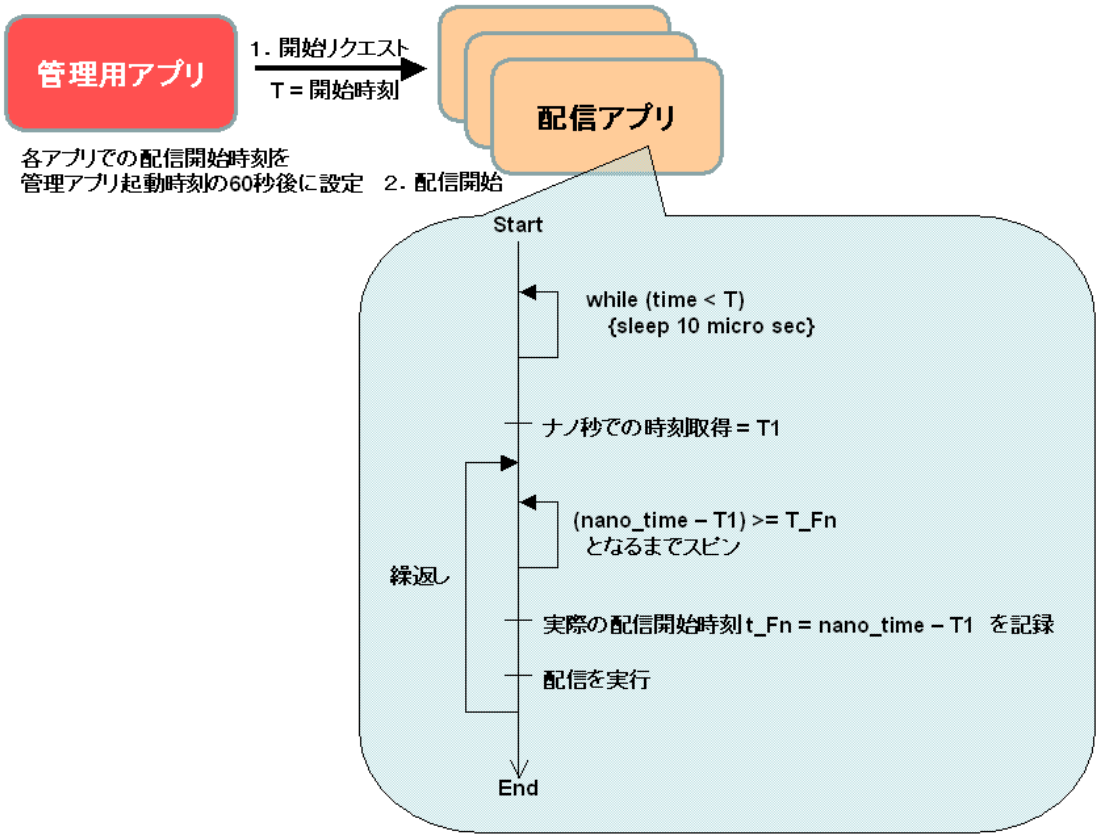

図 1-18　配信時の動作

<u>補足:時刻の精度について</u>

オリジナルのフィードデータの配信時刻はマイクロ秒の精度であり、このデータから Time Factor を使用してナノ秒の単位で配信開始時刻を設定した。配信アプリではある基準時刻からの経過時間取得に System.nanoTime()を使用することでコード上はナノ秒精度の時刻取得であったが、今回の検証環境では時間の分解能がマイクロ秒単位であったため実際の配信時刻合わせはマイクロ秒の精度であった。

(3) 測定結果

配信アプリ側で、設定された配信開始時刻から実際に開始した時刻との差（遅延時間）の結果を示す。平均値などの算出には、8 つの配信アプリ全ての測定データを使用した。

表 1-5 評価シナリオ3 配信側の遅延時間

サーバ数	Time Factor	遅延時間（micro sec）		
		平均	最小値	最大値
4	10	2,889	0	173,120
	20	15,532	0	682,682
8	10	1,433	0	105,757
	20	9,182	0	340,248

具体的に遅延が大きくなっている状況を確認するために、サーバ数 4 で時刻を 1/10 にした場合の、フィーダ番号1における配信の遅延時間を図 1-19 に示す。他のフィーダも同様である。また、時刻を 1/20 にした場合もほぼ同様だが、遅延の最大が数倍となっている。サーバ数が 8 の場合も、振る舞いはほぼ同様で、サーバ数 4 の場合の約半分ほどの遅延時間となっている。

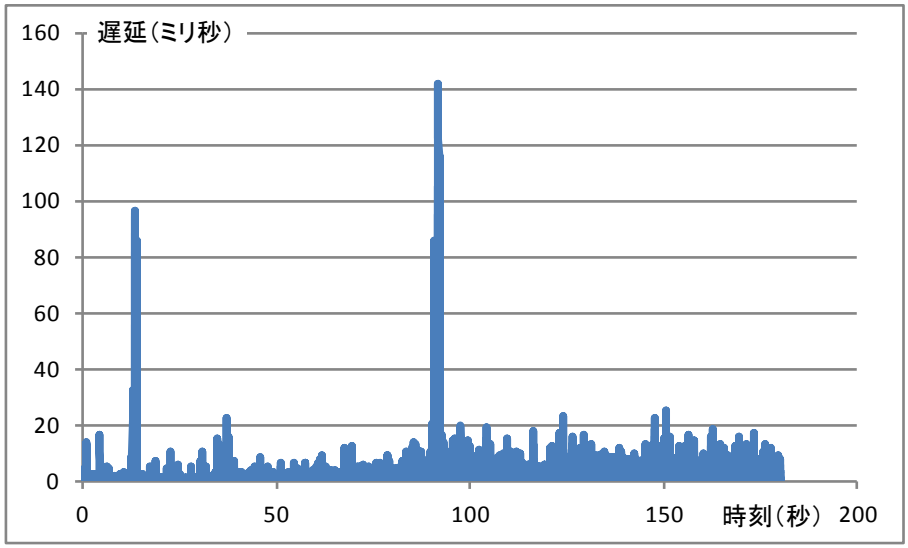

図 1-19 配信の遅延時間（サーバ数 4、時刻 1/10、フィーダ番号 1）

遅延が大きく発生している時刻の詳細を見てみる。時刻を 1/10 にした場合の最初の遅延のピークが発生している 13.5 秒前後の時点である。サーバ数 4 と 8 の場合を図 1-20 に示す。横軸が指定された時刻であり、縦軸が実際に配信された時刻である。階段状になっている部分では、配信に大きな遅延が生じたことを示している。サーバ数 4 の場合は遅延が大きかったが、サーバ数が 8 に増えるとかなり緩和されているのが分かる。一方、時刻を 1/20 にした場合、図 1-21 に示すように、元データの同じピークの場所では、サーバ数を 8 に増やしても大きな改善は見られない。遅延が発生する二つ目のピークは、時刻を 1/10 にした場合で 90 秒前後の時点である。一つ目の遅延のピークは、瞬間的な配信密度は高いが、二つ目の遅延のピークの方が、時間が長く、量も多いようである。そのため、図 1-22 に示すように、サーバ数を 8 にしても、影響がかなり残っているように見受けられる。1/20 にした場合も、図 1-23 に示すように、一つ目のピークに比べて改善が少ない。

10000/秒程度のイベント密度を 8 台のフィーダで配信する場合、1フィーダあたり 1250 イベント/秒、すなわち1イベントあたり 0.8 ミリ秒で処理される必要がある。しかし、Put の性能はおよそ 1 ミリ秒程度であるため、イベント密度が高まると遅延が生じる。フィーダの数を増やすと、遅延が緩和される可能性がある。

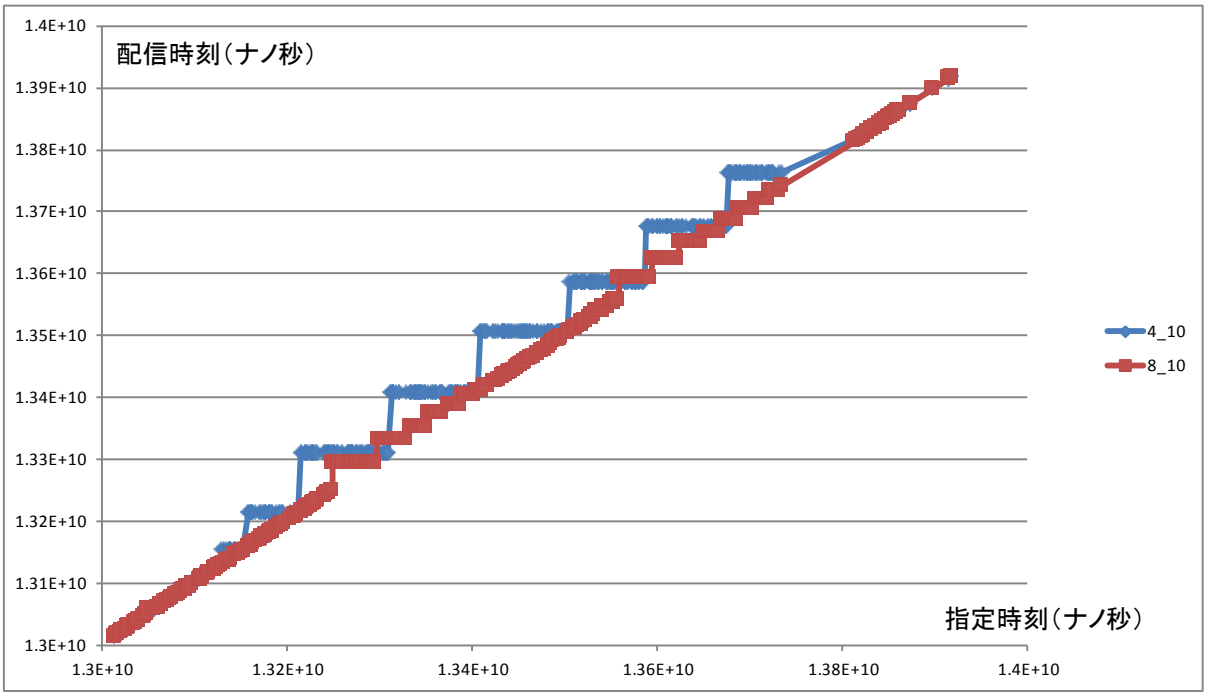

図 1-20 配信の遅延が大きい時点（一つ目のピーク、時刻 1/10、フィーダ番号 1）

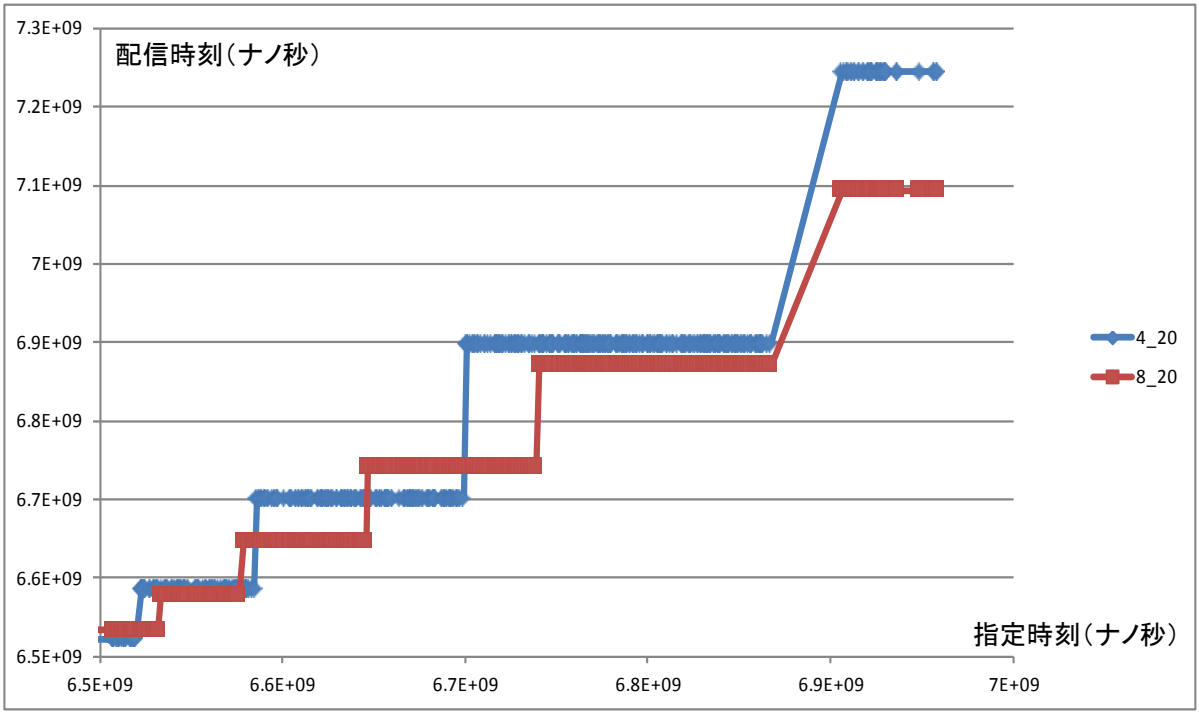

図 1-21 配信の遅延が大きい時点（一つ目のピーク、時刻 1/20、フィーダ番号 1）

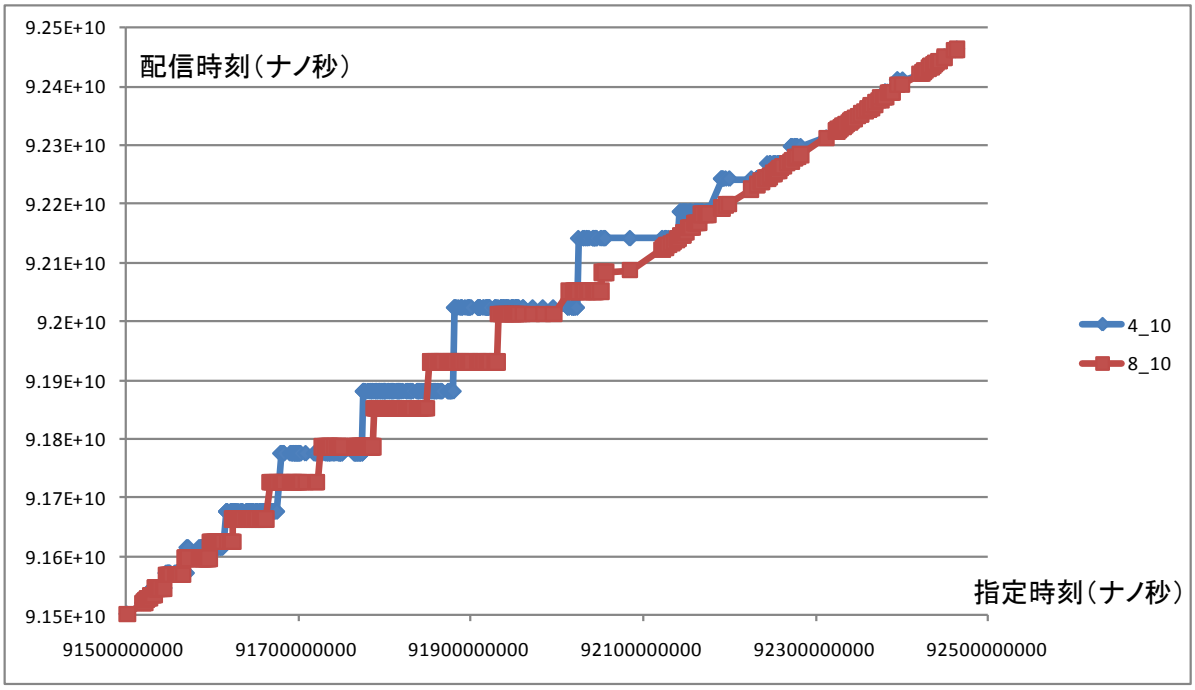

図 1-22 配信の遅延が大きい時点（二つ目のピーク、時刻 1/10、フィーダ番号 1）

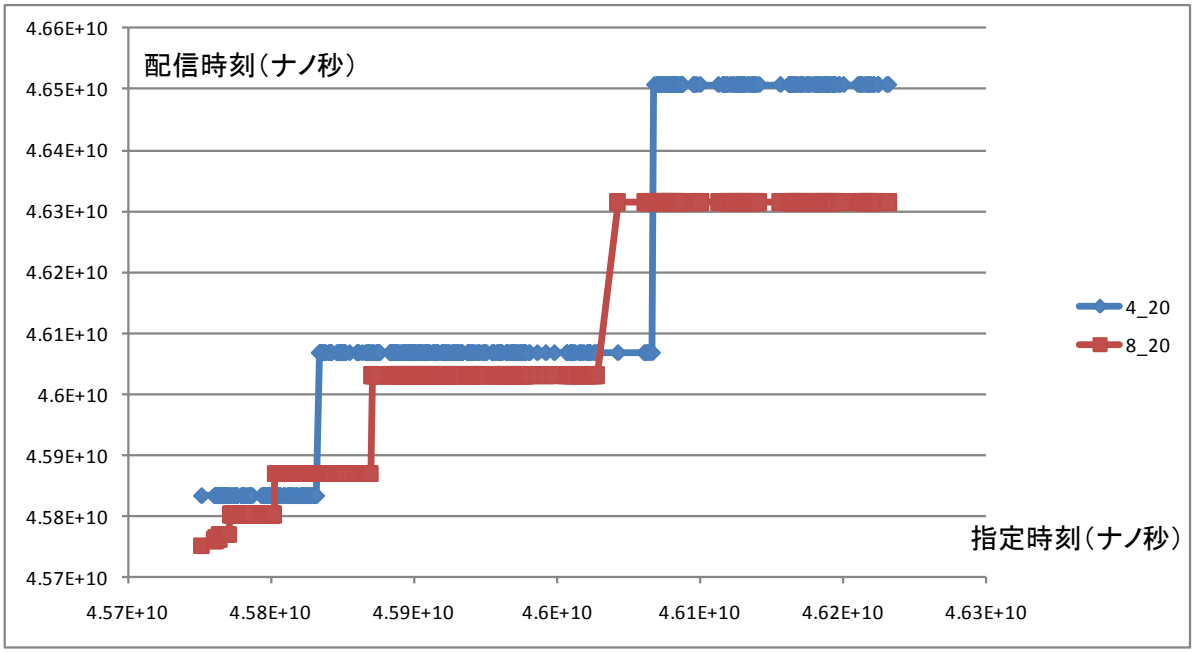

図 1-23 配信の遅延が大きい時点（二つ目のピーク、時刻 1/20、フィーダ番号 1）

データグリッド上で、配信後どのくらいの時間で書き込みが行われているかについては、ログが取得できないため確認することができない。

続いて、リスナによって受信される遅延について確認する。フィーダとリスナのイベント分布を図 1-24 と図 1-25 に示す。フィーダの密度はフィーダの送信時刻を sort し、2,000 イベント毎の所用時間から算出している。リスナ側の密度はリスナ側で計測した受信時刻を sort し、5,000 イベント毎の所用時間から算出している。ほとんどの配信データが一つもしくは複数のリスナイベントを生成するので、リスナの受信イベント数は配信した数より多くなっている。

時刻の短縮ファクタが同じ場合には、サーバ数が 4 でも 8 でもイベント密度はほとんど変化がない。時刻を 1/10 にした場合はフィーダのイベント密度のピークは 7,500 件/秒程度、1/20 にした場合は 10,000 件/秒を超えている。

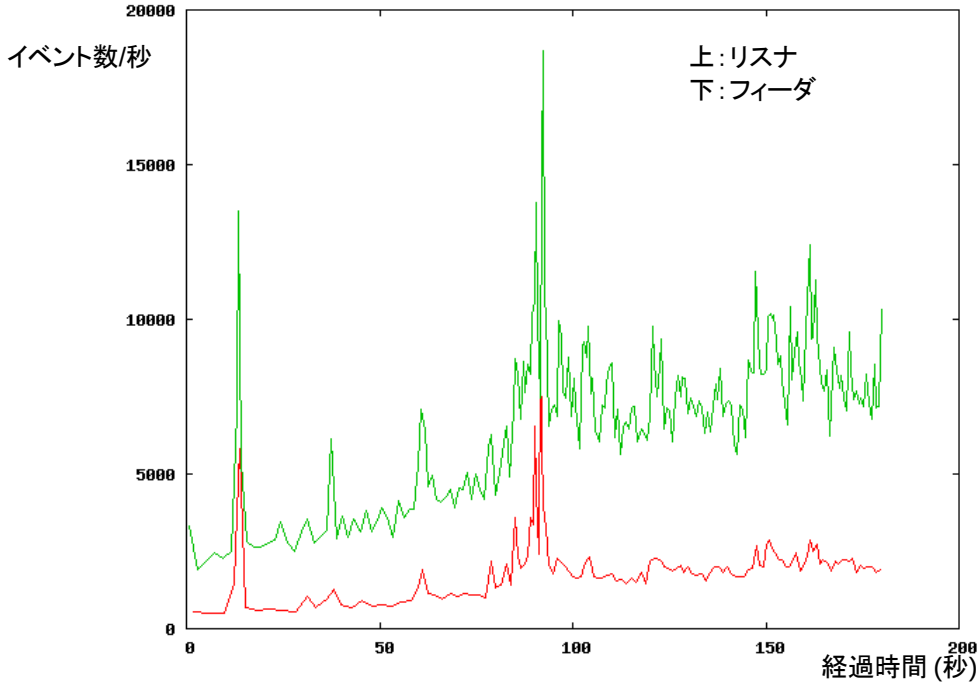

図 1-24 フィーダおよびリスナのイベント密度（サーバ数 4, 時刻 1/10 の場合）

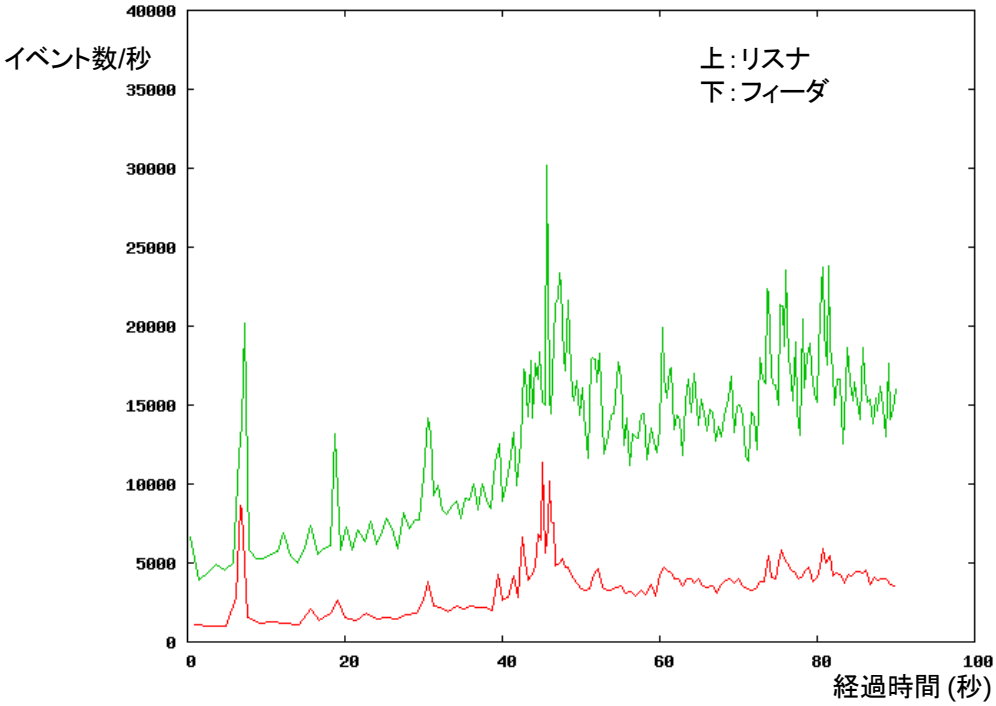

図 1-25 フィーダおよびリスナのイベント密度（サーバ数 4, 時刻 1/20 の場合）

フィーダが実際に配信を行い、リスナで受信されるまでにかかった時間を分析した。表 1-6 に遅延時間の概要を示す。配信側の遅延時間と同様に、時刻を短縮すると遅延が延び、サーバ数を増やすと遅延が短くなる傾向がある。標準偏差はおおまかに平均値と同等の大きさである。

表 1-6 受信側の遅延時間

サーバ数	Time Factor	遅延時間（micro sec）		
		平均	最小値	最大値
4	10	3,496	0	174,702
	20	14,699	0	688,177
8	10	2,222	10	114,193
	20	9,313	162	346,310

各時刻における平均遅延時間をサーバ数および時刻のファクタに変更したものをそれぞれ、図 1-26～図 1-29 に示す。遅延時間はフィーダでの受信時刻で sort し、5,000 イベント毎に遅延時間の平均を取っている。配信密度の高い場所で、フィーダで遅延が生じていたように、リスナでも遅延が生じている。リスナの遅延のピークとなっている時刻での遅延時間の分布をそれぞれ、図 1-30～図 1-33 に示す。遅延のピークとなっている時刻の近辺で 5,000～10,000 点の遅延時間の分布をプロットした。時刻を 1/20 にした場合では、遅延時間の大きいケースが増えている。また、サーバ数を 4 から 8 に増やすと遅延が半分ほどに減っている。秒 1 万件規模のリスナのイベントは、サーバ 4 台でも 0.2 秒以下の遅延でリッスンできるが、秒 3 万件規模のリスナのイベントは、サーバ 8 台でも最悪 0.4 秒程度の遅延が生じる。

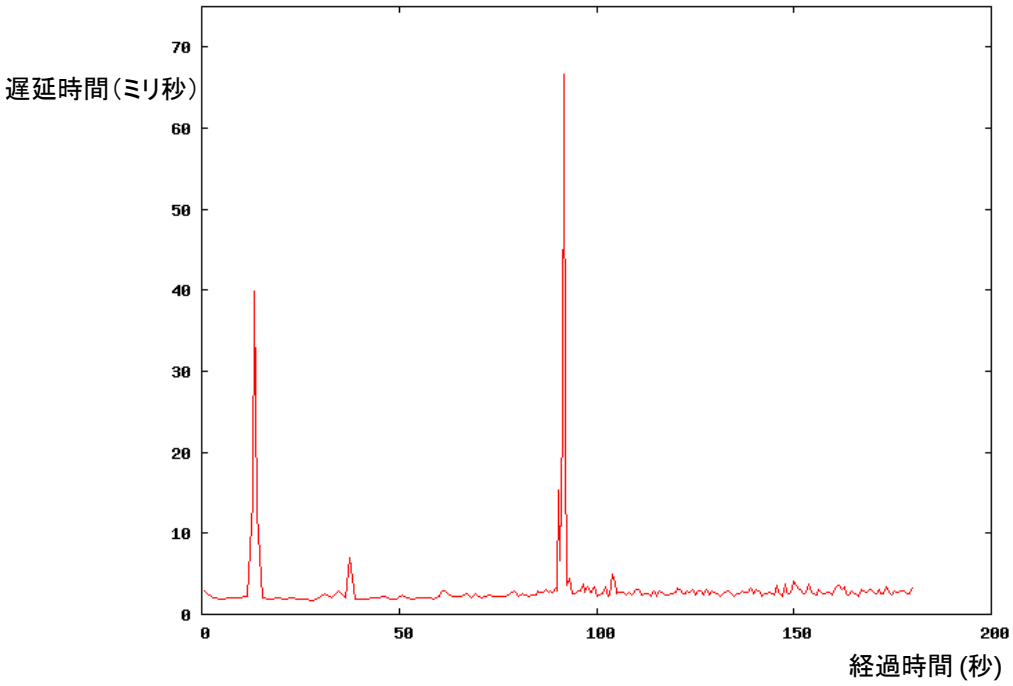

図 1-26 フィーダからリスナでの受信までの遅延時間（サーバ数 4, 時刻 1/10 の場合）

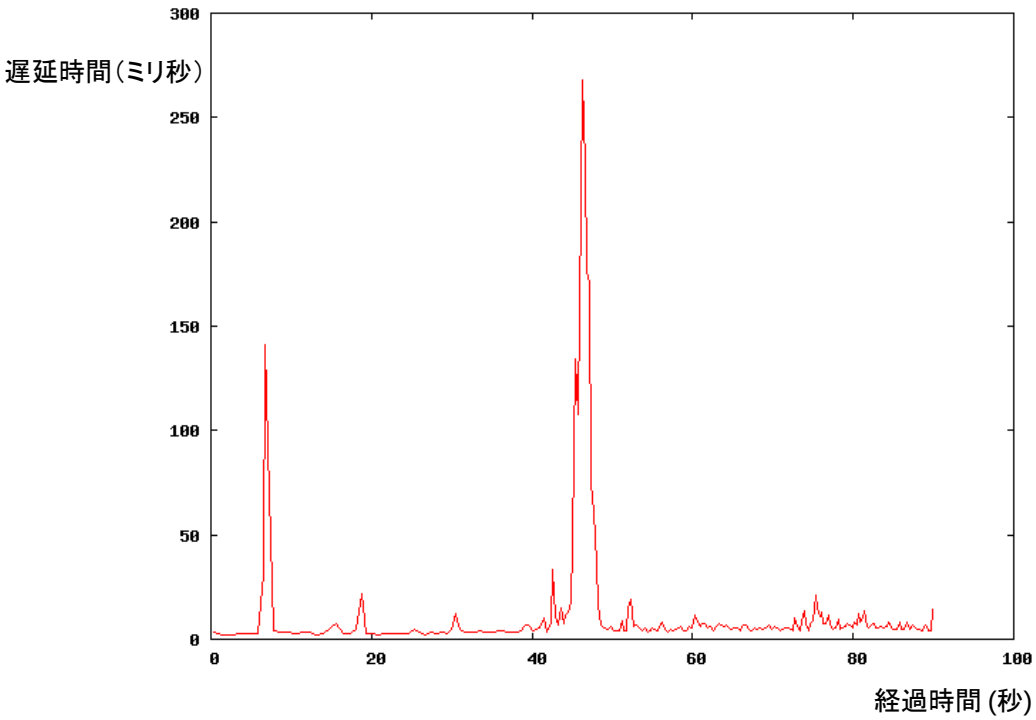

図 **1-27** フィーダからリスナでの受信までの遅延時間（サーバ数 **4,** 時刻 **1/20** の場合）

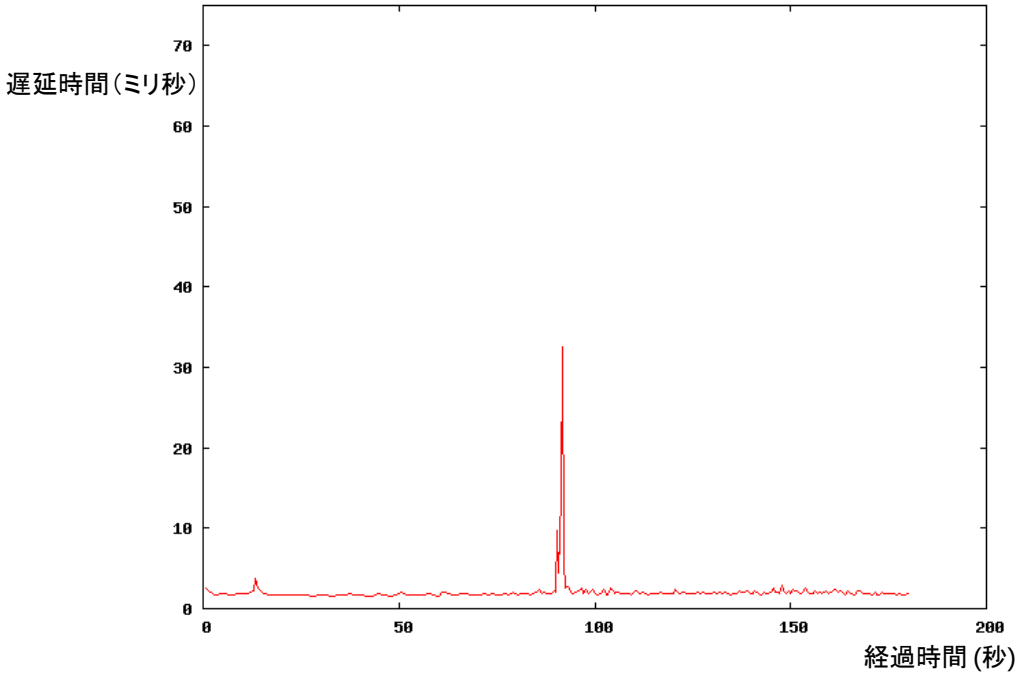

図 **1-28** フィーダからリスナでの受信までの遅延時間（サーバ数 **8,** 時刻 **1/10** の場合）

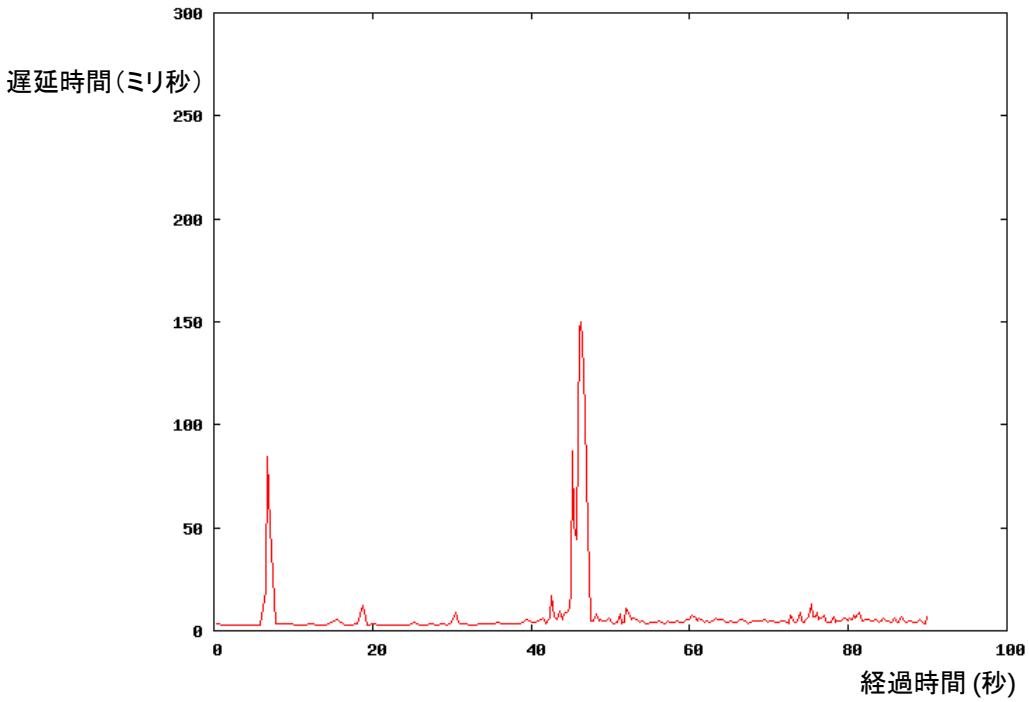

図 **1-29** フィーダからリスナでの受信までの遅延時間（サーバ数 **8**, 時刻 **1/20** の場合）

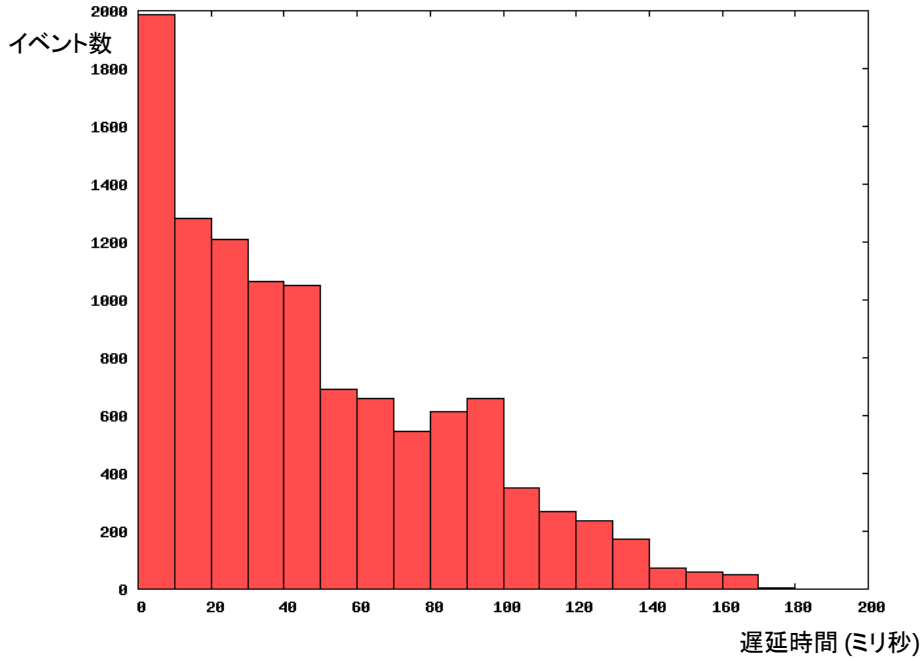

図 **1-30** リスナでの受信までの遅延時間の分布（サーバ数 **4**, 時刻 **1/10** の場合）

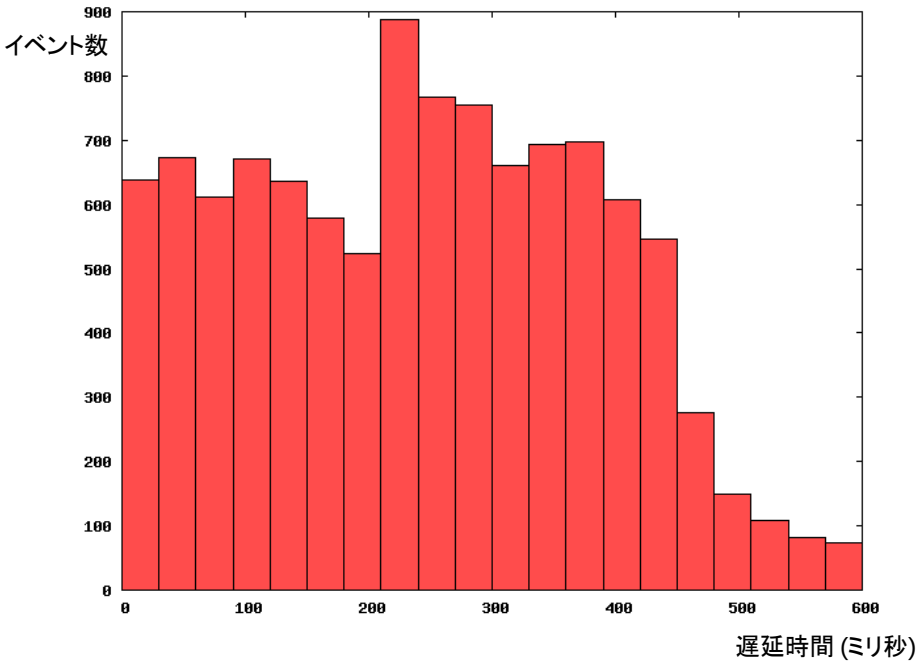

図 **1-31** リスナでの受信までの遅延時間の分布（サーバ数 **4,** 時刻 **1/20** の場合）

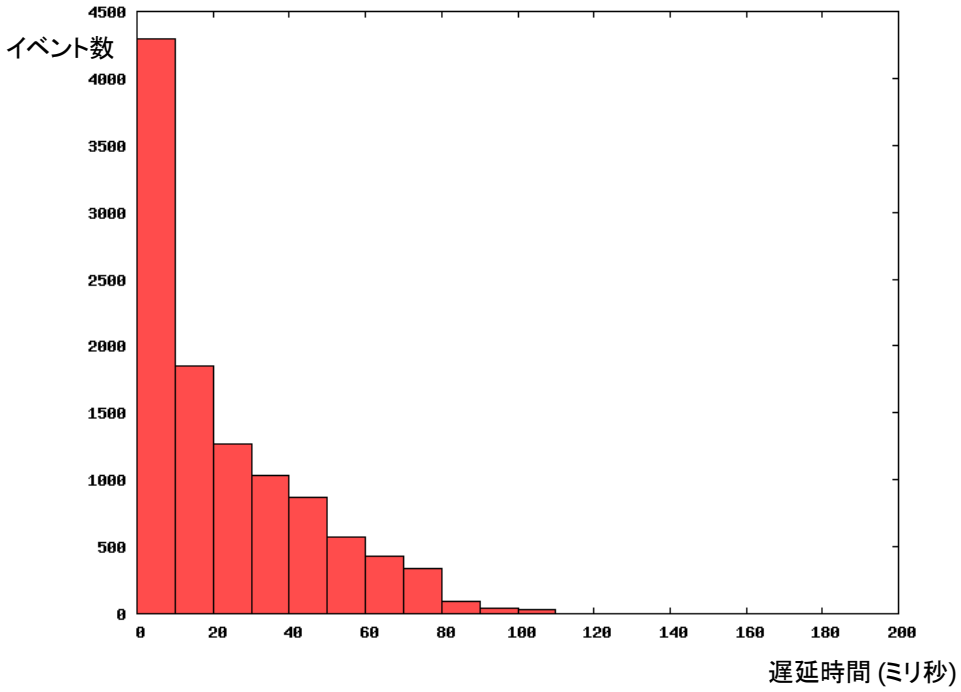

図 **1-32** リスナでの受信までの遅延時間の分布（サーバ数 **8,** 時刻 **1/10** の場合）

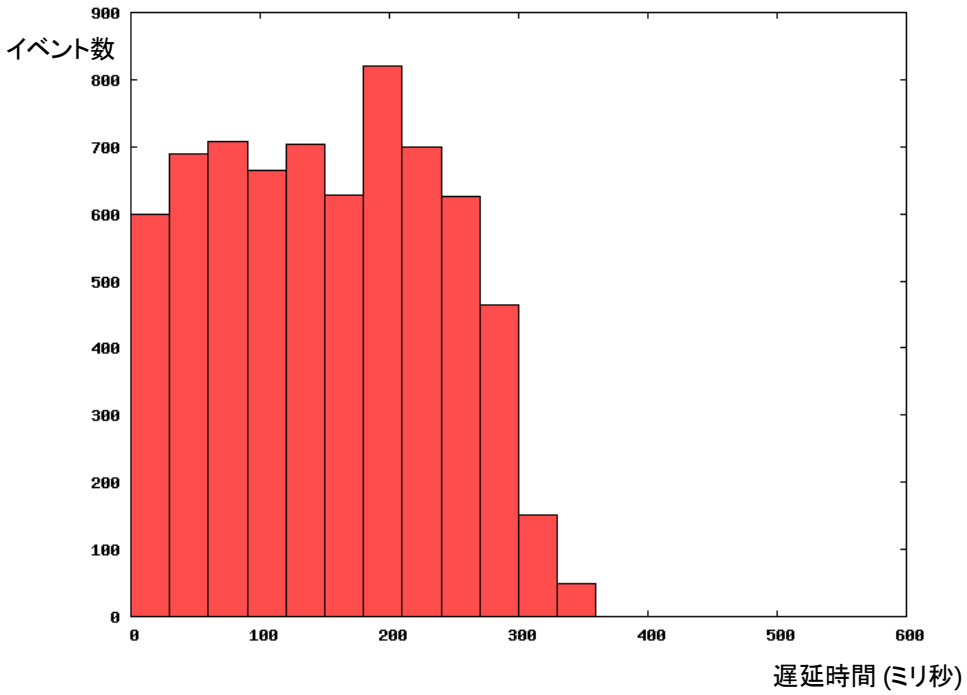

図 **1-33** リスナでの受信までの遅延時間の分布（サーバ数 **8,** 時刻 **1/20** の場合）