2005－OS－100（13）
2005／8／3

# 複数サブネット環境における自律的な故障検知機構

堀田勇樹[†]　田浦健次朗[†]　近山隆[†]

本稿では，信頼性の高い分散アプリケーションの記述を支援する故障検知機構の概要と，その実装を用いた評価および調査について報告する．以前提案したアルゴリズムにおいて問題が残されていた複数サブネット環境への適応部分を一部修正し，監視体系の構築速度および故障検知の信頼性を改善する．3 クラスタ 297 ノード上で動作させたところ，数秒で自律的に監視体系が構築されることが確認された．また実環境におけるデータが乏しい故障検知パラメータに関する調査を行い，実用における限界点について評価を行った．

# Autonomous Failure Detectors in Multi-subnet Environments

Yuuki Horita,[†] Kenjiro Taura[†] and Takashi Chikayama[†]

This paper presents our failure detectors for supporting to implement reliable distributed applications. We revise our algorithm described in [20] with respect to adapting to multi-subnet environments so as to improve speed of creating monitoring relations and accuracy of failure detection. When we ran our failure detectors on 297-nodes in 3-clusters, in which the underlying connectivity is partially limited, the monitoring relations were autonomously constructed within several seconds. We investigated system parameters, such as heartbeat interval ($T_{hb}$) and timeout ($T_{to}$), by using our implementation, and evaluated their practical limits.

## 1. はじめに

近年，クラスタの普及や高性能のネットワークの整備により，大量にかつ広域に分散した計算資源を利用する機会が増えてきている．分散環境で動作するアプリケーションは，構成するプロセス数の増加に伴い，プロセスやノードの故障，ネットワーク障害などに遭遇する可能性が高まるため，故障が発生した場合に適切な対処（自己修復など）ができるよう設計されていることが望まれる．その際どの構成要素が故障したのかを迅速に検知することが不可欠となる．

リモートプロセスの故障は自ずと通知されるわけではない．TCP プロトコルでは，接続先のプロセスがクラッシュすると接続の切断通知が届けられるため故障を検知できることがあるが，ハードウェアレベル（ノード）の故障やネットワーク障害の場合は通知を受け取らないため検知することができない．heartbeat（keep alive）などメッセージの反応の有無を適宜チェックすることによって判断するしかない．そのため，その監視単位で常時通信が発生することになり，全対全や集中管理型などの単純な監視方式では，プロセス数の増加に従いたちまち無視できない負荷（CPU, 通信量）が発生する．こうした背景により，従来からスケーラビリティを考慮した様々な故障検知手法が提案されてきた[6),11),17),18)]．

一方，耐故障性を持つ分散アプリケーションを記述するための枠組みもメッセージパッシングライブラリを中心に発達してきている[1),2),7)]．しかしながら，それらの基盤となる故障検知は，TCP の切断通知[2)] やスケーラビリティの欠ける単純な heartbeat[1),7)] に依存してしまっている．これは効率的な故障検知機能を提供する実装が存在していないことが大きく影響している．MDS[5)]，NWS[19)]，Ganglia[14)] のような資源管理システムを利用することも可能であるが，それらのシステムは資源情報の管理・提供が本来の目的であるため耐故障アプリケーションの基盤としてはあまり使い勝手が良くない上，分散アプリケーションを実行する環境で実際に稼動しているとは限らない．耐故障性を持つ分散アプリケーションを作成，またはその支援機構を構築するためには，故障検知部分も自分で実装しなければならないというのが現状となっている．この手間が，耐故障分散アプリケーションを作成することの障壁を高くしている要因の一つであり，従来から提案されてきた手法と現実に適用されている手法との乖離を生んでいる原因であると考えられる．

そこで本研究では，ユーザレベルで容易に利用でき

† 東京大学
University of Tokyo

る実用的な故障検知機構を実装・提供することを目的としている．我々は，すでに[20] で，高い効率性・信頼性を持つ実環境に即した故障検知システムを提案し，実験によりその有効性を示した．しかし，複数サブネット環境への適応部分において構築時間や故障検知の正確性の点で課題が残されていた．本稿では，[20] で述べたアルゴリズムを一部修正し，それらの問題の改善を図る．また，実際に利用する際に問題となる，heartbeat 間隔や timeout までの時間などの故障検知機構のパラメータに関する調査を行い，故障検知速度とシステム負荷や正確性における均衡点について評価する．

本稿の構成は以下のようになっている．2 節で関連研究について述べ我々の研究との対比を行う．3 節では，[20] で提案した故障検知機構におけるアルゴリズムを整理，修正する．4 節で実験・評価を行い，最後にまとめる．

## 2. 関 連 研 究

FT-MPI[7] は，代表的なメッセージパッシングの規格である MPI[15),16)] を耐故障並列計算用に拡張した通信ライブラリで，アプリケーションのプロセスが故障したときに通信関数などを通じてエラーを通知する仕組みを備えている．しかし，FT-MPI は基本的に接続の切断通知を頼りに故障を検知している．heartbeat で故障を監視する watchdog と呼ばれる故障検知モジュールも用意されてはいるものの，全対全の heartbeat というスケーラビリティのない単純な実装になっている．また，我々は故障検知機構を独立した機能として提供するため，様々な用途に利用することができる．FT-MPI などの耐故障通信ライブラリに組み込むことも可能である．

故障検知機構としては，従来よりスケーラビリティを考慮した手法がいくつか提案されてきた．[17] では，監視プロセスを階層的に構成することにより，heartbeat による負荷を分散させている．これは資源管理システム[5),14),19)] で主に用いられている手法で，トポロジーに沿って全体の情報を一箇所に効率的に収集することができるという利点がある．しかし，監視プロセスが故障するとシステムが機能しなくなるという問題がある．冗長化により信頼性を高めることも可能であるが，情報の一貫性やフェイルオーバーの必要性などプロトコルが複雑になることは避けられない．また，起動時や環境の変化による再構成時に適切に監視体系を構築するのは，ユーザにとって大きな手間となる．

Gossip[18] は，各プロセスの heartbeat 情報を定期的にランダムな相手に送信しあうことにより，噂が広まるようにして全体に heartbeat を伝達する手法である．$O(\log n)$ 回送信しあうことにより非常に高い確率で全体に情報が伝達されることが証明されているため，プロセス数が増加しても故障検知時間にはあまり影響を与えない．各プロセスの機能は対称的であり自律的に監視体系を構築するので，複数故障に高い耐性を持ち，手動設定を必要としない．だが，通常の heartbeat による手法よりも故障検知までに時間を要するという点や，heartbeat 情報はプロセス数に応じてサイズが大きくなるため $O(n^2)$ の通信量が発生するという点などにおいて問題がある．

SWIM[6),11)] では，故障検知機能と故障情報を伝達する機能を分離することにより，そのような Gossip の問題点を改善している．各プロセスは，定期的にランダムにプロセスを選び ping-pong をする．もしもある一定時間以上 ping に対する返事がなかったら，そのプロセスの故障を疑い，その情報を全体に伝達する．各プロセスは高い確率で他プロセスから ping-pong の要求を受けるため，故障が起きた場合は Gossip に比べ短い時間で検知される．また ping-pong に用いるメッセージは小さくてよいため，一プロセスあたりの通信量は一定である．しかし，この手法は全対全で通信可能な環境を想定しており，接続性が制限されているような一般的な環境では正常に動作しない．

Gossip protocol を応用した Reliable multicast の分野では，multicast するための決定的なパスを我々の故障検知機構と近い手法で構築するものもある[9),10),13)]．だが，いずれも目的が異なることからそれをそのまま故障検知機構として利用するには適していない．

[3] で示されているように，正確に全ての故障を検知することは不可能であり，パケットロスや遅延などによって度々誤検知が発生する．故障検知速度を維持しつつ正確性を高めるために，ネットワークの状況に適応する故障の検知手法も盛んに研究されている[4),8)]．また[12] は，一つの故障検知機構で複数のアプリケーションの要求に対応するため，正常／故障の 2 値ではなく，連続した故障確率を提供するという手法を提案している．これらの研究は一つ一つの監視の仕方を定めるものであり，我々の監視体系の構築手法とは独立している．そのため，これらの監視手法を我々の故障検知機構に組み込むことも可能である．本稿では監視手法については扱わない．

## 3. 故障検知機構

この節では，我々の故障検知機構の詳細について述べる．3.1 節で同一サブネット内で動作する基本的なプロトコル[20] を説明し，3.2 節では，複数サブネット

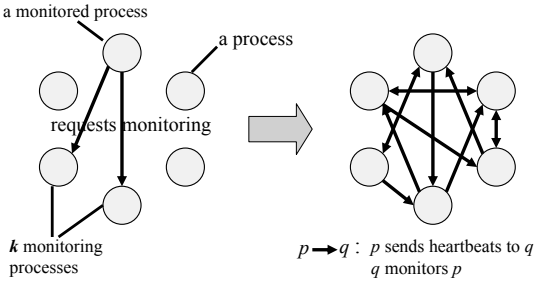


**図 1** Creating Monitoring Relations

環境に拡張したプロトコルについて述べる．[20] においても複数サブネット環境で動作するアルゴリズムを提案したが，本稿ではそれを改め新しい手法を提案する．

### 3.1 基本プロトコル

我々の故障検知機構では，システムに参加しているプロセスの間で分散した監視体系を自律的に構築する．各プロセスは自分を監視するプロセス（以下，監視プロセス）が $k$ 個存在するように動作する．もしも自分の監視プロセスの数が $k$ 個に達していなかったら，参加プロセスの中からランダムに選び，自分の監視を依頼する（図 1 左）．逆に $k$ 個を超えていた場合は，余分なだけ監視関係を破棄する．各プロセスが $k$ 個のプロセスによって監視されるようにすることで，監視による負荷を軽減すると共に故障検知の信頼性を高めている．

これにより，プロセスが故障したときには $k$ 個の監視プロセスによって検知される．しかし，その故障情報は全プロセスが知る必要があるため，故障検知のために構築した監視体系を利用して全体に伝達する．監視体系は，最終的に（図 1 右）のようなグラフ構造になる．これを無向グラフと見立てた場合に全体として非連結となる確率は，プロセス数 $n$ が増加していくと $k \geq 2$ であれば少なくとも $O(n^{-k^2-2})$ よりも急速に 0 に収束していくことが証明されている．また，$n$ が小さいときの非連結確率も $k$ が大きくなるに従って小さくなり，故障による一時的な破損の影響を考慮しても，$k$ を適切に（例えば 4 や 5）設定することで，非常に高い連結性を維持することができる．そこでこの性質を利用して，監視体系に沿った flooding により故障情報を伝達する．flooding では全てのエッジに対して最悪 2 回のメッセージが流れることになるが，この監視グラフのエッジ数は $kn$ であるため，せいぜい $2kn$ のメッセージで済む．また，伝達のパスが冗長化されているため，信頼性の高い情報伝搬を実現できる．

### 3.2 複数サブネット環境への拡張

#### 3.2.1 従来のアルゴリズム

前節では，各プロセスが互いに直接通信できるような環境を想定していたが，現実の環境ではファイアウォールや NAT などの存在により異なるネットワーク間の通信は制限されていることが多い．そのような環境で基本プロトコルを適用すると，各ネットワークで閉じた監視体系を構築する確率が高く，監視体系が全体として連結しない恐れがある．例えば，2 つの計算機クラスタがあり，両者はゲートウェイにあたる一つの計算機同士でしか通信できないような環境を考える．一クラスタの計算機数を $m$ とすると，監視の依頼はランダムに行われるため，ゲートウェイノード上のプロセスがクラスタ間を結ぶ監視関係を築く確率は $\frac{{}_{m-1}C_{k-1}}{{}_{m}C_{k}} = \frac{k}{m}$ となる．これでは両クラスタ共に計算機数 $m$ が大きかった場合に，クラスタ間には監視関係が張られない可能性が高くなってしまう．

この問題を回避するために，我々は以前隣接プロセス情報を用いて全体の連結性を保つ手法を提案した[20]．隣接情報を元に全対全で通信可能な部分集合（クリーク）を推定し，各プロセスは自分が所属している各クリークに対して $k$ 個の監視プロセスが存在するように依頼する．するとクリーク内では基本プロトコルと同じような監視体系が構築されるので，前節で述べた非連結確率の議論が成立し，全体が非常に高い確率で連結される．

しかし，この手法は実用面においていくつか問題を抱えている．

- ネットワークトポロジーを考慮せず接続性のみを情報として監視体系を構築するため，異なるサブネット間で接続が大量に張られる可能性がある
- LAN の接続（同一サブネット内）と信頼性の低い WAN の接続（異なるサブネット間）を区別せず同じパラメータで監視するため，正確性（もしくは故障検知速度）が低下する
- NAT やファイアウォールの設定によっては，連結性が非対称となりクリークとして扱えない場合がある
- 各プロセスは隣接プロセス情報を取得する必要があるため，監視体系の構築に時間を要する

そこで我々は，複数サブネット間で監視体系を連結させるためのアルゴリズムを次節で述べるように修正する．

#### 3.2.2 修正アルゴリズム

修正アルゴリズムでは，自分の所属しているサブネット内部と外部で異なるアルゴリズムを適用する．プロセスがどのサブネットに所属しているかは，ioctl や ifconfig で取得可能なブロードキャストアドレスの比較によって知ることができるので，事前に情報を与える必要はない．各プロセスは同一サブネット内に対

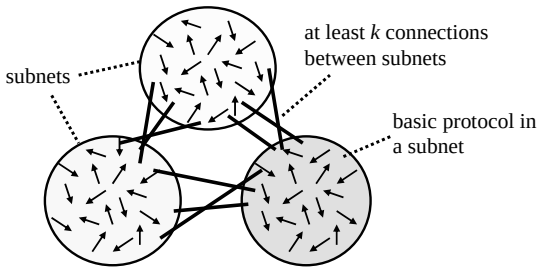


**図 2** Monitoring Relations in multi-subnets

しては基本プロトコルに従って監視体系を構築し，異なるサブネットに対しては各サブネットから少なくとも $k$ 本の接続が張られるようにする（図 2）．サブネット間の接続は，両端のプロセスが互いに監視する．3.1 節で述べたように，サブネット内では非常に高い確率で連結した監視グラフが生成される．この手法はその各サブネット間を $k$ 本のエッジで接続していることになるので，監視体系の連結性は全体として損なわれない．

図 3 は，その監視体系を構築するアルゴリズムである．サブネット間の接続数を制御するためには，接続情報をサブネット内で共有しなければならない．そのため，他のサブネットに所属するプロセスと接続が確立したら，監視グラフ上にその情報を flooding してサブネット内のプロセスに伝達する（add_wconn）．接続情報は同期していないので，実際には $k$ 本よりも多くの接続が張られるかもしれないが，余分な接続は後で間引くことも可能である．

なお，図 3 のアルゴリズムでは，簡単のため，プロセス情報（$N(c)$）やサブネット情報（$C$）を既知のものとして扱っているが，実際には動的に収集される．これは，故障を検知した時と同様で，検知した新規プロセスの情報を flooding で通知することによって達成される．そのため我々の実装では，プロセスが動的に参加することも可能である．この際必要となるのは，システムに参加中の一プロセスの hostname（IP アドレス）と listen port だけであり，このように手動で与える情報を最低限に抑えることにより，ユーザは手間をかけず容易にシステムを利用することができる．

また，この手法は従来のアルゴリズムと比較して以下のような特徴を持つ．

- 異なるサブネット間の接続数は $O(kc^2)$（$c$: サブネット数）であり，プロセス数に依存せず小さく抑えられる
- 信頼性の低いサブネット間の接続は緩やかに監視し，サブネット内では積極的に監視することにより，故障検知の正確性と速度を保つことができる

/* a connection between $p$ and $q$ is represented as $(p, q)$
 * a subnet that $p$ belongs to is represented as $c_p$ */
$S_m$ : a set of the procs. that $p$ monitors within $p$'s subnet
$S_h$ : a set of the procs. that monitor $p$ within $p$'s subnet
$C$ : a set of the subnet
$N(c)$ : a set of the procs. belonging to a subnet $c$
$W(c)$ : a set of inter-subnet connections between $c_p$ and $c$
$H_w$ : a set of the procs. in other subnets that $p$ monitors

**Initially :**
  for $c$ in $C$ :
    $W(c) = \phi$
    if $c = c_p$ :
      < use basic protocol >
    else :
      call ensure_connectivity($c$)

**procedure flood_within_subnet($m$, $q$) :**
/* *forward $m$ on all conns. within $p$'s subnet, except to its source $q$* */
  for $q'$ in $(S_m \cup S_h) - \{q\}$
    send $m$ to $q'$

**procedure add_wconn($p'$, $q'$, $q$) :**
/* *called when the proc. learns a new inter-subnet conn. between two procs. $p$' and $q$' from a proc. $q$* */
  $W(c_{q'}) = W(c_{q'}) \cup \{(p', q')\}$
  call flood_within_subnet(*wconn_info*($p$, $p'$, $q'$), $q$) /* notify within this subnet */

**procedure ensure_connectivity($c$) :**
/* *called so that at least $k$ conns. to a subnet $c$ will be established* */
  /* there may be less than $k$ conns. to a subnet $c$ */
  if $|W(c)| < k$ :
    pick $q$ randomly from $N(c)$
    send *monitor_req*($p$) to $q$

**Recv *monitor_req($q$)* :**
  send *monitor_ack*($p$) to $q$
  if $c_q = c_p$ :
    < use basic protocol >
  else : /* from another subnet */
    $H_w = H_w \cup \{q\}$ /* start monitoring this conn. */
    call add_wconn($p$, $q$, $p$)

**Recv *monitor_ack($q$)* :**
  if $c_q = c_p$ :
    < use basic protocol >
  else :
    $H_w = H_w \cup \{q\}$ /* start monitoring this conn. */
    call add_wconn($p$, $q$, $p$);

**Recv *wconn_info*($q$, $p'$, $q'$) :**
/* *notified of a new inter-subnet conn. between $p'$ and $q'$ from $q$* */
  if $(p', q') \notin W(c_{q'})$ :
    call add_wconn($p'$, $q'$, $q$)

**$T'_{to}$ has passed since $p$ sent *monitor_req($p$)* to $q$ where $c_q \neq c_p$ :**
/* *$p$ could not connect to $q$* */
  call ensure_connectivity($c_q$)

**図 3** Modified Algorithm for creating monitoring relations (we are omitting the detail of our basic protocol, so if necessary, refer to [20])

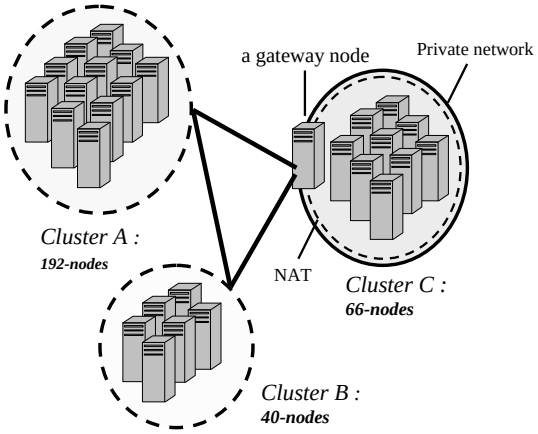


**図 4** Experimental Environment

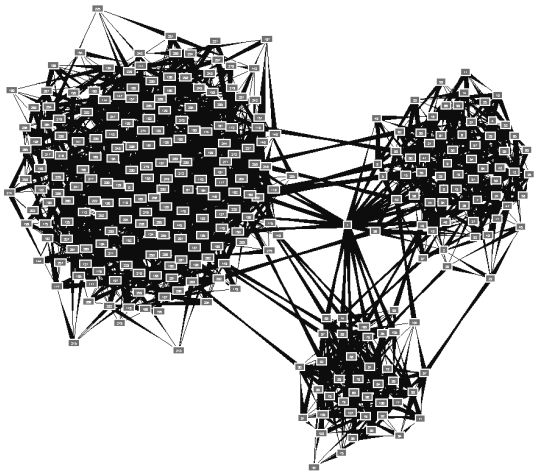

**図 5** Monitoring Relations in 3 clusters

- 確立された接続を双方向から利用するため，NAT やファイアウォールなどによって片側からのみ接続が許されている場合でも対応できる
- 事前に情報を必要としないので，速やかに監視体系を構築できる

従って，従来のアルゴリズムにおける問題点は解消され，システムの迅速な構築および信頼性の高い故障検知が可能となる．

## 4. 実験・評価

我々は 3 節で述べたアルゴリズムの修正に伴い，故障検知機構を改めて実装した．この節では，監視体系の構築速度やパラメータに関する実験について述べる．監視体系の信頼性や他手法との比較に関しては[20] を参照いただきたい．

実験環境としては以下のものを用いた．

- Cluster-A (SMP) at Hongo :
  Xeon 2.4GHz (70 nodes), Linux2.4.20
  Xeon 2.8GHz (122 nodes), Linux2.6.9
- Cluster-B (single) at Hongo :
  Xeon 2.4GHz (31 nodes), Linux2.4.27
  Xeon 3.06GHz (9 nodes), Linux2.4.27
- Cluster-C (SMP) at Kashiwa :
  Xeon 2.4GHz (66 nodes), Linux2.4.20

クラスタ間の接続性は図 4 のようになっている．Cluster-A と Cluster-B は，各ノードがグローバル IP アドレスを持ち外部と自由に通信できる．Cluster-C は NAT になっており，ゲートウェイノードは外部と自由に通信できるが，他のノードは内部から外部への通信のみ可能となっている．

### 4.1 動作確認

図 5 は，3 つのクラスタ計 297 ノード上でプロセスを立ち上げたときに構築された監視体系である（ただし $k = 5$）．クラスタ内では基本プロトコルに従って密な監視関係を構築し，クラスタ間では接続数が小さく抑えられていることが確認できる．ただし現時点の実装では，余分な接続が張られても放置しているため，$k$ 本よりも多くの接続が張られている．Cluster-C のゲートウェイノードは，Global IP と Private IP を持ち，前者のサブネットに所属しているノードは他に存在しないため，各サブネットと $k$ 本の接続が張られている．しかし，同一プロセスに $k$ 本の接続を張ることは本来の目的である信頼性への効果は薄く，サブネット数が増えると大量の接続が張られることになり望ましくない．そのため，将来的にはプロセスあたりのサブネット間の接続数に何らかの上限値を設けて制限する必要があると考えられる．

### 4.2 構築速度

我々の故障検知機構は，ユーザレベルで利用されることを想定しているため，故障検知機能を利用する分散アプリケーションと同時に立ち上げられることが予想される．そのため，できる限り早く機能を提供できるように，速やかに監視体系を構築する必要がある．これは動的に監視対象を追加された場合も同様である．そこで監視体系の構築時間を測定し，評価を行う．

まず，Cluster-A の 160 ノード上で立ち上げたときの監視体系の構築時間を測定した．各プロセスには初期情報としてある一プロセスの情報（ホスト名とポート番号）を与えた．つまり，全てのプロセスはシステム起動時にそのプロセスに対して接続を張ることになる．ここで構築時間とは，監視関係が新たに張られなくなり，かつ全プロセスが参加プロセス全ての情報を取得するまでの時間と定義する．図 6 は 30 回の試行の結果である．比較結果として，あらかじめプロセス情報を与えておき全対全で接続を張るプログラムの実

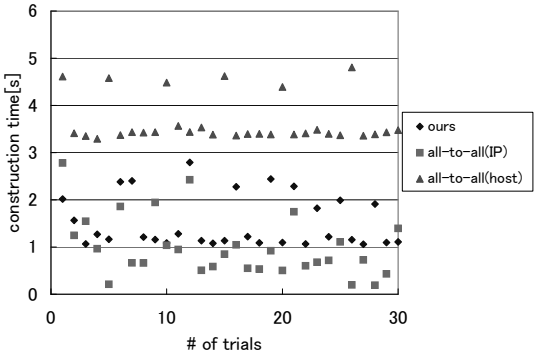


**図 6** Construction Time in a cluster (160-nodes)

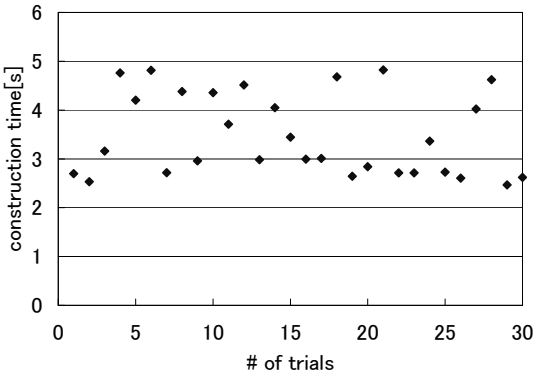


**図 7** Construction Time in 3-clusters (297-nodes)

行時間も掲載している．all-to-all(IP) は IP アドレスを与えた場合，all-to-all(host) はホスト名を与えた場合である．

我々の故障検知機構は，大半が 1 秒台で構築されているものの，若干ばらつきのあることが観察できる．これは一プロセスに接続要求が集中し，connect に失敗することがあるためであると考えられる．その現象は全対全での接続においても表れている．我々の手法ではプロセス情報を自律的に収集する必要があるので，IP アドレスが与えられて単に接続するだけの all-to-all(IP) には及ばないが，DNS で IP アドレスを引く作業が必要な all-to-all(host) よりは良い結果が出ている．これは同時に大量のプロセスからアクセスを受けるため DNS がボトルネックとなっていることを意味しているわけだが，事前に全てのプロセス情報を与える場合ホスト名を記述させるのが自然であり，同じような現象が生じることが推測される．それに対し自律的にプロセス情報を取得する場合は，DNS を引くのは初めだけで済むためあまり影響はない．

次に，前節の動作確認と同じ環境で測定した構築時間の結果を図 7 に示す．約 3 〜 5 秒程度で構築されており，一部接続性が制限されている環境でもそれほど構築速度が劣化していないことがうかがえる．これは

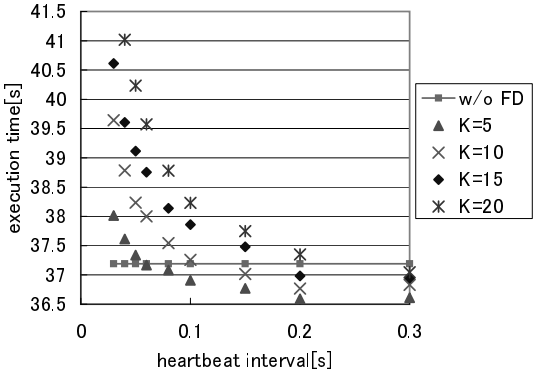


**図 8** System Overhead (execution time of a fibonacci program (N=46))

各プロセスが他プロセスに接続を試行する数が，プロセス数やサブネット数に従って増えることがないことに起因する．そのため，システムの規模が大きくなっても，構築時間は急激に劣化することはない．

なお，この構築時間は，故障検知機構の監視体系の収束を表す指標であり，全体が接続されていれば，それ以前より故障検知機能を提供することができる．よって，実用的な構築速度の範囲に収まっているといえる．

### 4.3 パラメータの調査

故障検知機構には必ず heartbeat 間隔（$T_{hb}$）と故障と判断するまでの timeout 時間（$T_{to}$ ($> T_{hb}$)）の少なくとも 2 つのパラメータが存在する．この 2 つのパラメータは故障検知の速度・正確性，システム負荷を左右する．$T_{hb}$ を短くすると，故障検知時間が短くなるのに対し，システムの負荷は大きくなる．$T_{to}$ は長いほど，誤検知の可能性を小さくすることができるが，故障検知時間が長くなる．こうしたトレードオフの関係により，適切にパラメータを設定するのは難しい．理論上やシミュレーションで故障検知の速度と正確性の解析をしている研究は存在するが[4),18)]，実際の挙動やシステム負荷に関するデータは乏しく不明な点が多い．そこで，我々の実装において $T_{hb}$ を変化させシステム負荷および heartbeat の挙動について調査を行った．

#### 4.3.1 システム負荷

監視依頼数 $k$ 及び heartbeat 間隔 $T_{hb}$ を変化させたときの，故障検知機構によるオーバヘッドをフィボナッチプログラム（実行時間約 37 秒）を用いて測定した．我々の故障検知機構はランダム性により実際にはプロセスによって他プロセスの監視数が異なるが，ここでは平均的な値を取り $k$ 個の監視をしている場合を扱った．実験環境としては，single CPU の Cluster-C を用い，測定は 2.4GHz の計算機で行った．

結果を図 8 に示す．故障検知機構を background で走らせていると通常の実行よりも速くなるという現象が発生した．これは通常起こりえないことであるが原因は不明であり，現在調査中である．そのため基底値が曖昧なので，ここでは実行時間の上昇率に着目して評価を行う．

heartbeat 間隔が短くなるに従って，実行時間は急激に増加していっており，監視依頼数 $k$ が大きいほどその上昇率は高い．これは頻繁に発生する通信やコンテキストスイッチによるオーバヘッドが表面化してきたためであり，これより監視の負荷分散が迅速な故障検知の実現にとって重要であることが確認される．通常監視依頼数 $k$ は 5 程度に設定されることを想定しているが，先にも述べたように実際には監視数においてばらつきがあり，中には数十のプロセスに監視を依頼されるものもある．ある一定以上の監視数を超えたら監視の依頼を断るようにすることで，ある程度分散させることは可能だが，完全に均一化することはできない．それを考慮すると，ユーザのアプリケーションに与える影響を小さく抑えるためには $T_{hb} = 0.1[s]$ 程度が限界であるといえる．これはもちろん計算機の性能によって変わってくるものなので，実用的には $T_{hb} \geq 0.2$ ぐらいが妥当であると考えられる．

#### 4.3.2 heartbeat の挙動

我々の故障検知機構では，信頼性の低いサブネット間の接続はあまり頻繁に監視せず，同一サブネット内で迅速に故障を検知する．これは，一箇所で故障が検知されれば，その情報は flooding によってすぐに通知されるので，敢えて信頼性の低い接続で検知する必要がないためである．よって，同一サブネット内での heartbeat を調査の対象とする．

Cluster-A 190 ノード上で $k = 5$ に設定した故障検知機構を $T_{hb} = 0.1[s]$ と $T_{hb} = 0.2[s]$ でそれぞれ約 $2.5 \times 10^5$ heartbeat 時間（約 25000 秒，約 47000 秒）走らせ，監視プロセスが heartbeat を受信する間隔について調べた．図 9 と図 10 にそれぞれの結果を示す．これは，全 heartbeat（約 $190 \times 2.5 \times 10^5$ 個）のうち受信間隔が理論値 $(= T_{hb})$ よりも 0.02[s] 以上長かったものを全てプロットしたものである．

いずれの結果も，heartbeat が頻繁に遅延を受けていることを示している．主な理由としては，他のプロセスの影響で heartbeat を送信するスレッドまたはメッセージを受信するスレッドへのスケジューリングが遅れるためであると考えられる．開始直後に大量の遅延が発生しているが，全て起点として用いたプロセスが関連しており，全プロセスから通信が集中したために処理が間に合わなかったためであると推測される．

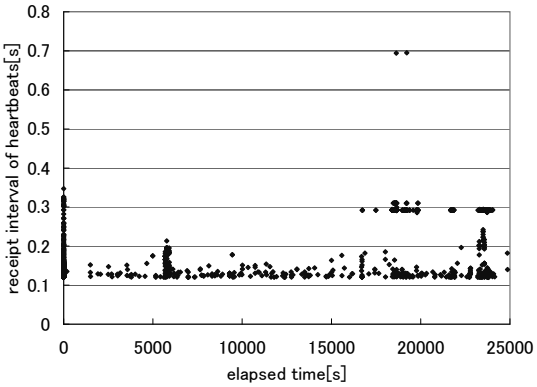


**図 9** Behavior of heartbeats ($T_{hb} = 0.1[s]$)

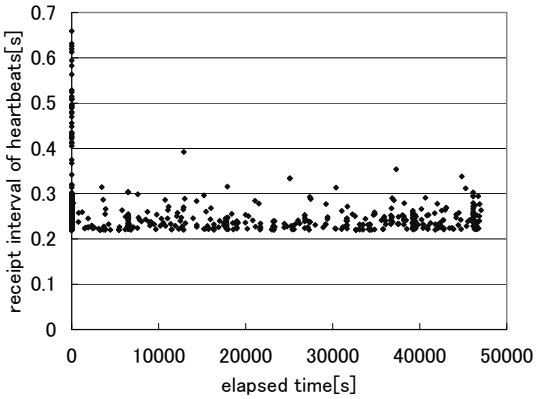


**図 10** Behavior of heartbeats ($T_{hb} = 0.2[s]$)

また，図 9 の後半部において大きな遅延が発生しているが，この時間帯には同一クラスタ上で他プロセスが走っていたことが確認されている．詳細は不明だが，遅延の大きさからパケットロスによる再送制御によるものではないかと考えている．

このように heartbeat は数 100ms 程度の遅延は頻繁に受けるものと考えるべきであり，$T_{to}$ は少なくともそれより大きめに設定する必要があるといえる．ネットワークの状態によっては予測不能な遅延が発生する可能性もあり，ユーザ側の正確性に対する要求度合に応じて設定することが求められる．

## 5. おわりに

本稿では，耐故障分散アプリケーションを支援する故障検知機構の設計・実装を行った．実用面で問題が残されていた以前の提案手法を修正し，システムの迅速な構築を可能にし，故障検知の信頼性を向上させた．本手法に基づいた実装を 3 クラスタ 297 ノード上で動作させたところ，数秒で自律的に監視体系が構築されることが確認された．また故障検知パラメータに関する調査を行い，実用における限界点について評価した．

今後の課題としては，API の実装，ネットワークの状況に適応する監視手法の検討，ネットワーク分断時への対応などが考えられる．

## 参 考 文 献


1) R. Batchu, Y. Dandass, A. Skjellum, and M. Beddhu. MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware. *Cluster Computing*, pp. 303–315, October 2004.
2) Aurelien Bouteiller, Franck Cappello, Thomas Herault, Geraud Krawezik, Pierre Lemarinier, and Frederic Magniette. MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. 2003.
3) T. D. Chandra and S.Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, pp. 225–267, March 1996.
4) Wei Chen, SamToueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 2002.
5) K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proc. of Intl. Symposium on High Performance Distributed Computing*, 2001.
6) Abhinandan Das, Indranil Gupta, and Ashish Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *Proc. of Intl. Conf. on Dependable Systems and Networks (DSN'02)*, pp. 303–312, June 2002.
7) G. E. Fagg and J. J. Dongarra. Building and using a Fault Tolerant MPI implementation. *International Journal of High Performance Applications and Supercomputing*, 2004.
8) C. Fetzer, M. Raynal, and F. Tronel. An adaptive failure detection protocol. In *Proc. 8th IEEE Pacific Rim Symposium on Dependable Computing (PRDC-8)*, pp. 146–153, December 2001.
9) Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulie. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Proc. Third Intl. Workshop Networked Group Communication*, pp. 44–55, November 2001.
10) Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulie. Hiscamp: self-organizing hierarchical membership protocol. Proc. SIGOPS European Workshop, September 2002.
11) Indranil Gupta, Tushar D. Chandra, and German S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proc. of 20th Annual ACM Symp. on Principles of Distributed Computing*, pp. 170–179, 2001.
12) Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama. The $\varphi$ Accrual Failure Detector. In *Proc. of the 23rd IEEE International Symposium on Reliable Distributed Systems(SRDS-23)*, October 2004.
13) Meng-Jang Lin and Keith Marzullo. Directional gossip: Gossip in a wide area network. In *European Dependable Computing Conference (EDCC)*, pp. 364–379, 1999.
14) Matthew L. Massie, Brent N. Chun, and DavidE. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. Technical report, February 2003.
15) Message Passing Interface Forum. MPI: A Message Passing Interface Standard, June 1995. http://www.mpi-forum.org/.
16) Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, July 1997. http://www.mpi-forum.org/.
17) Paul Stelling, Ian Foster, Carl Kesselman, Craig Lee, and Gregor von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, Vol. 2, No. 2, pp. 117–128, 1999.
18) R. van Renesse, Y. Minsky, and M. Hayden. A Gossip-Style Failure Detection Service. In *Middleware '98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 55–70, 1998.
19) Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems, 15(5-6)*, pp. 757–768, October 1999.
20) 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. 先進的計算基盤システムシンポジウム (SACSIS2005), May 2005.