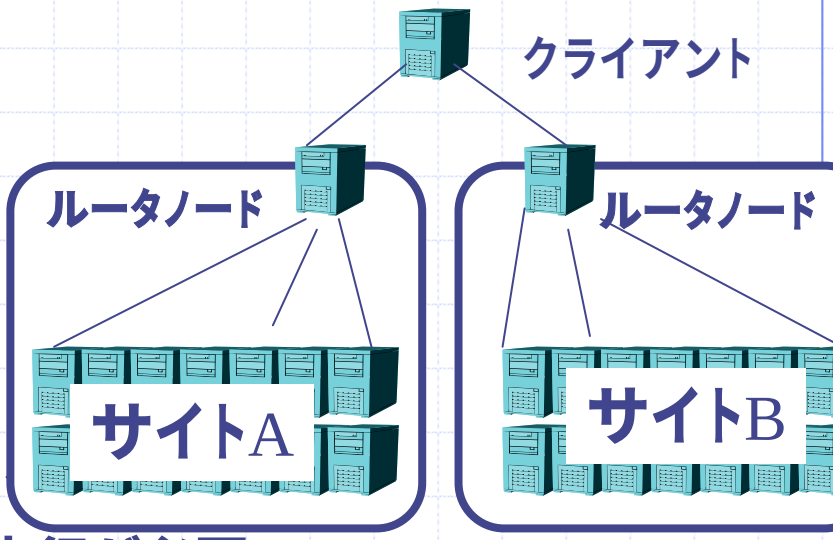
Javaによる階層型グリッド環境Jojoの設計と実装

中田秀基 (産総研、東工大) ,
松岡聡(東工大、国情研),
関口智嗣(産総研)

背景



- グリッドとクラスタの普及
  - グリッドノードとしてのクラス
    - 複数のクラスタにまたがった実行が必要
- 問題点1: クラスタ各ノードはプライベートアドレス
  - 既存のツールでは十分な活用は難しい
    - Ex. MPICH-G2 、C.f. NXProxy [Tanaka]
- 問題点2: クライアントのスケーラビリティ
  - 数百ノードに対してスケールするかどうか不明

⇒グリッドの階層化

目的

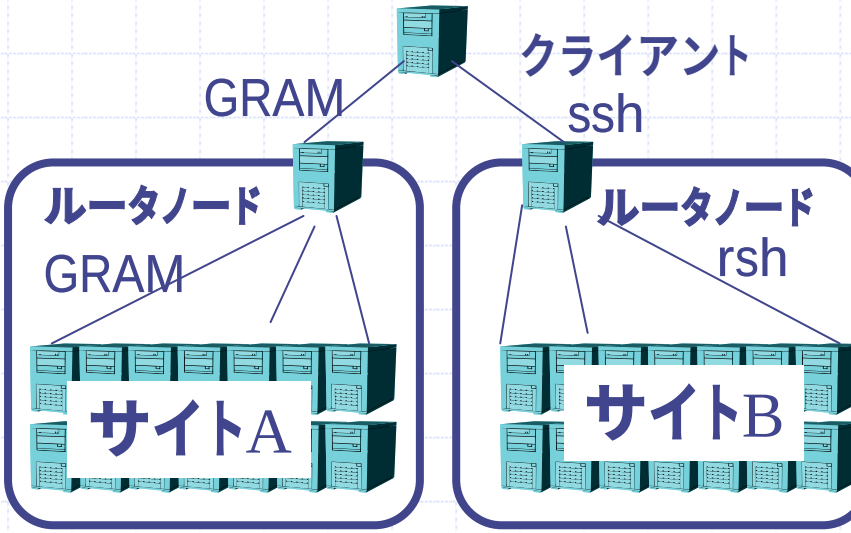
- 階層構造を持つグリッド環境に適したプログラミング環境を提供
  - クラスタノードがプライベートアドレスであっても活用可能
  - 階層化することでスケーラビリティを向上

Javaを用いたグリッド向けプログラミング環境

Jojo

Jojoの特徴(1)

- ツリー状の通信を行うので、プライベートアドレスでも問題なく使用が可能
  - 再帰的な起動
  - 起動プロトコル:Globus GRAM、ssh、rsh
    - 混在可能



Jojoの特徴(2)

- 階層構造を持つノードのそれぞれで実行されるプログラムを記述
  - 階層構造を意識したプログラミングが可能
  - 各ノード上のプログラムは自分の近傍ノード(親、子、兄弟)と通信
- Jojoのシステムプログラムやユーザのプログラムを動的にダウンロード
  - システムのバージョン違いなどのトラブルを未然に防ぐ
  - クラスタの各ノードにはJava VMが実行できる環境だけがあればよいのでインストールの必要がない

プログラミングモデルへの要請

- 簡潔で直感的
  - 実行時のセットアップも考慮
- 柔軟なメッセージパッシング
  - スレッドを利用した通信と計算のオーバラップによるレイテンシの隠蔽

  ⇒複雑な非同期通信を容易に記述できることが不可欠

関連研究

- JavaRMI (Sun)、Horb(ETL)
  - 分散オブジェクトに対する同期メソッド呼び出し
  - 並列実行はスレッドと組み合わせて実現
  - オブジェクトを別途起動、登録する必要
  - コード(オブジェクト)の配置が煩雑
- JPVM,mpiJava etc.
  - Send, recv によるメッセージパッシングモデル
  - 簡潔なSPMDモデル
  - Send,Recvだけで複雑な同期を記述するのは難しい

Jojoのプログラミングモデル

- 各ノードに1つの代表オブジェクトが起動
  - Codeクラスのサブクラスを実行
  - SPMD的
- Codeがオブジェクト単位で送受信を行う
  - 受信は基本的にハンドラメソッドで行う
  - 送信に対する返信は頻出するので特別扱いしていくつかの通信モードを用意
    - ブロッキング呼び出し
    - フューチャーオブジェクトやコールバックオブジェクトを指定するノンブロッキング呼び出し

API (1) Code

```
abstract class Code{
  Node [] siblings;    /** 兄弟ノード */
  Node [] descendants; /** 子ノード */
  Node    parent;      /** 親ノード */
  int     rank;  /** 兄弟の中での順位 */
  /** 初期化 */
  public void init(Map arg);
  /** 本体の処理 */
  public void start();
  /** 送信されてきたオブジェクトの処理 */
  public Object handle(Message mes);
}
```

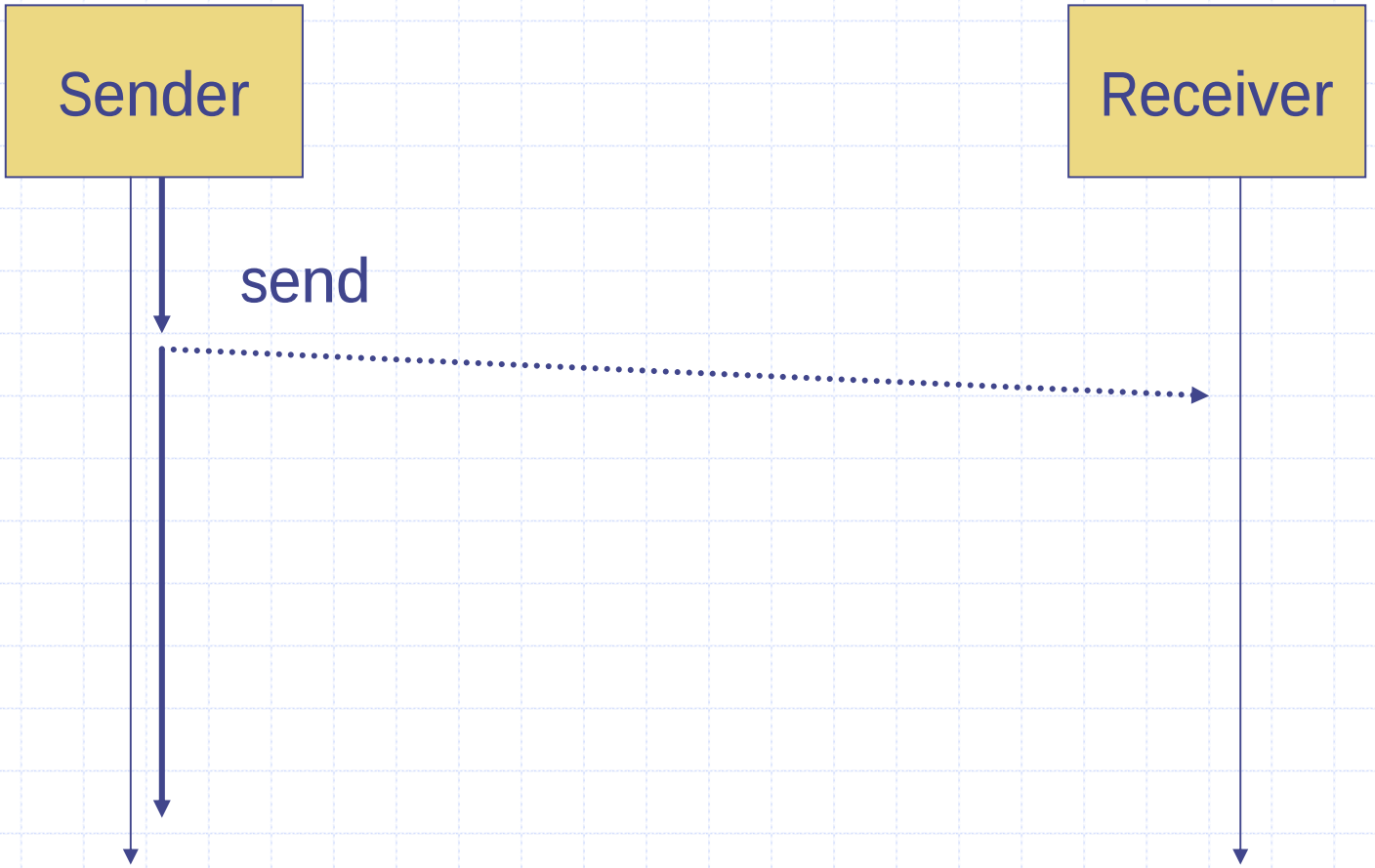
API (2) Node

```
public interface Node {
  /** non-blocking send; do not wait for reply */
  void send(Message msg) ;

  /** blocking call; wait for reply */
  Object call(Message msg) ;

  /** non-blocking call; do not wait for reply.
   *  returns Future to synchronize the reply. */
  Future callFuture(Message msg) ;

  /** non-blocking call; execute unnable */
  void callWithContext(Message msg, Context context) ;
}
```

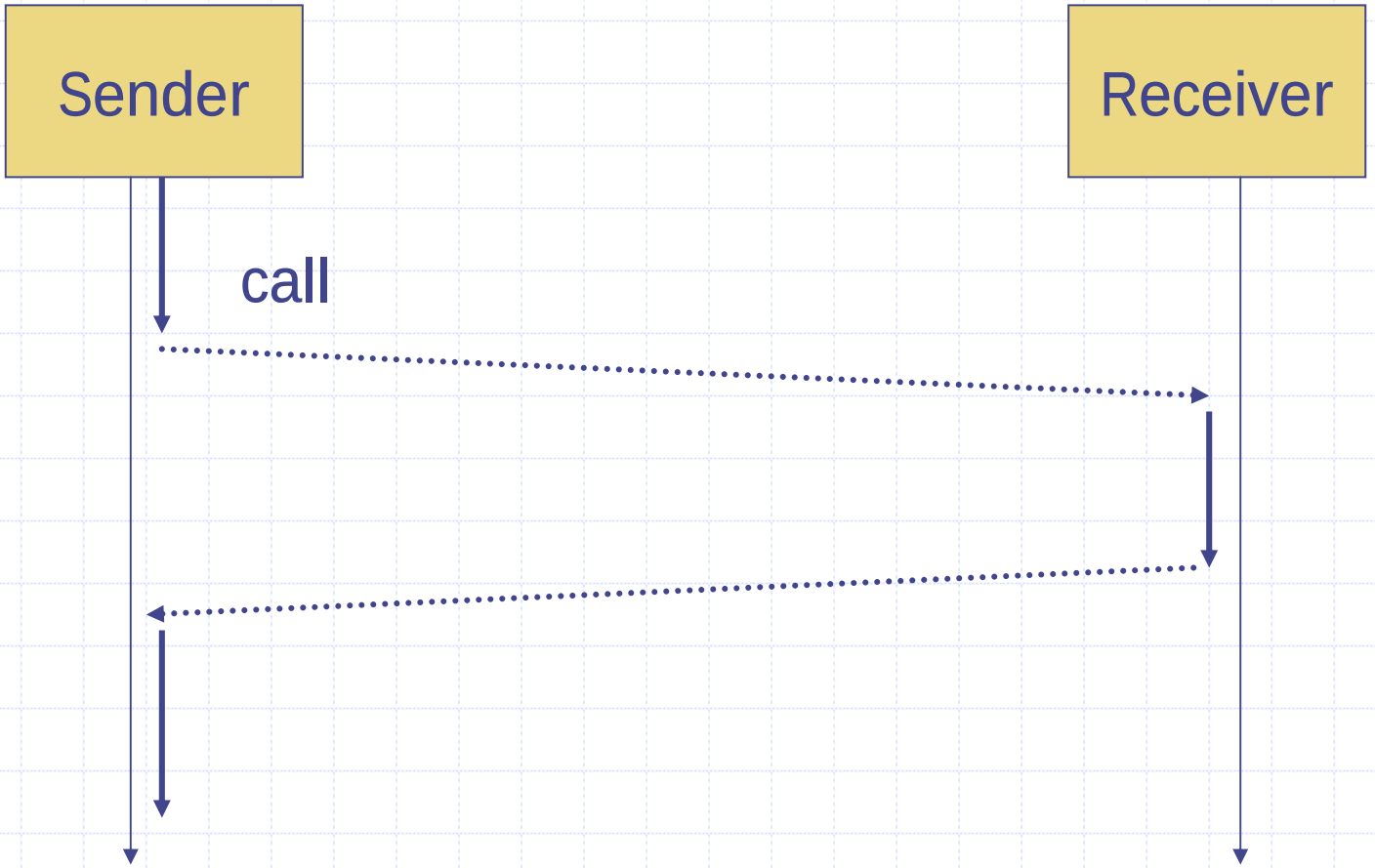
API (3) Message

```
public class Message implements Serializable{
  /** message id */
  public int tag;

  /** message contents */
  public Serializable contents;
}
```

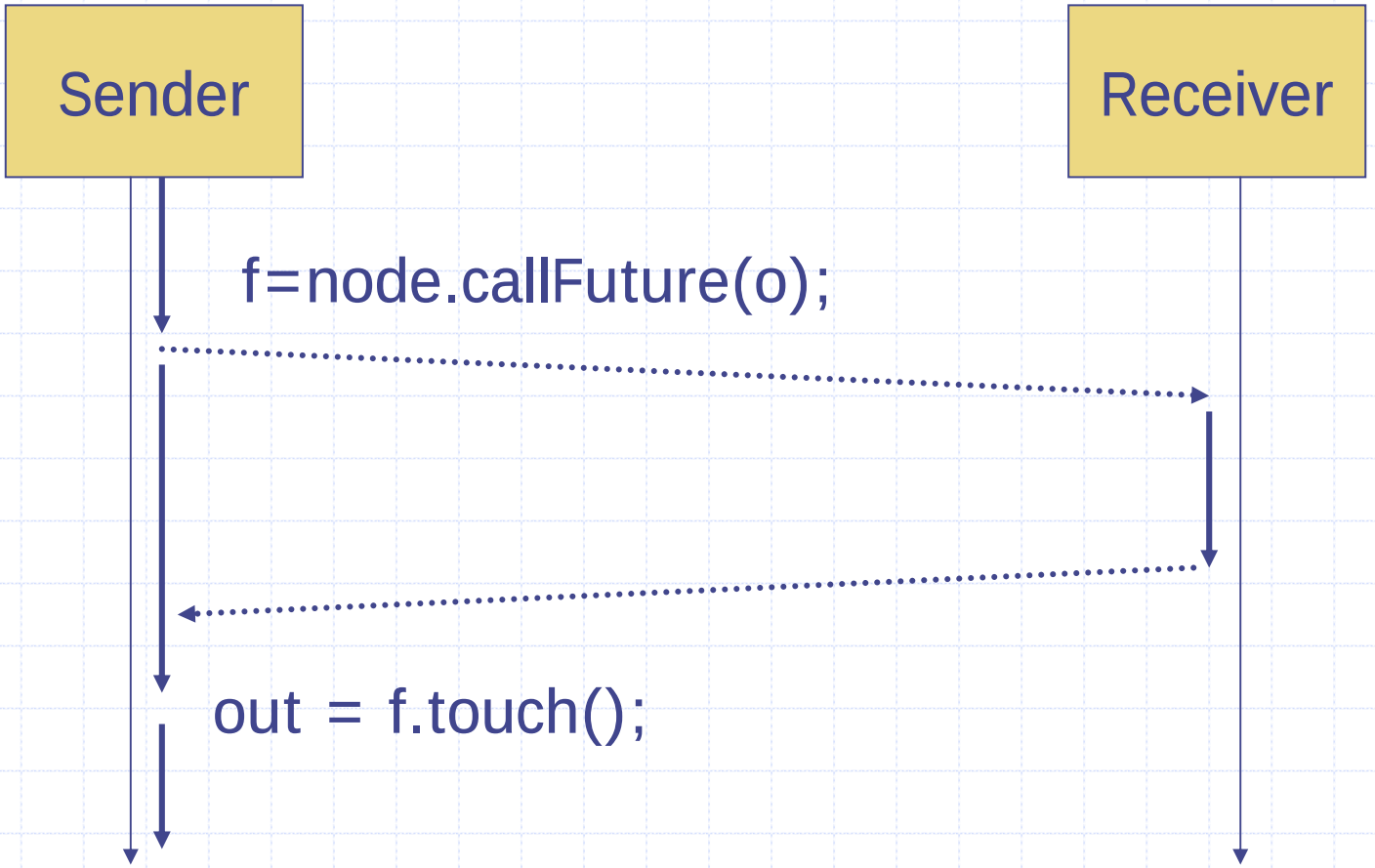
通信モード(1) 送信のみ



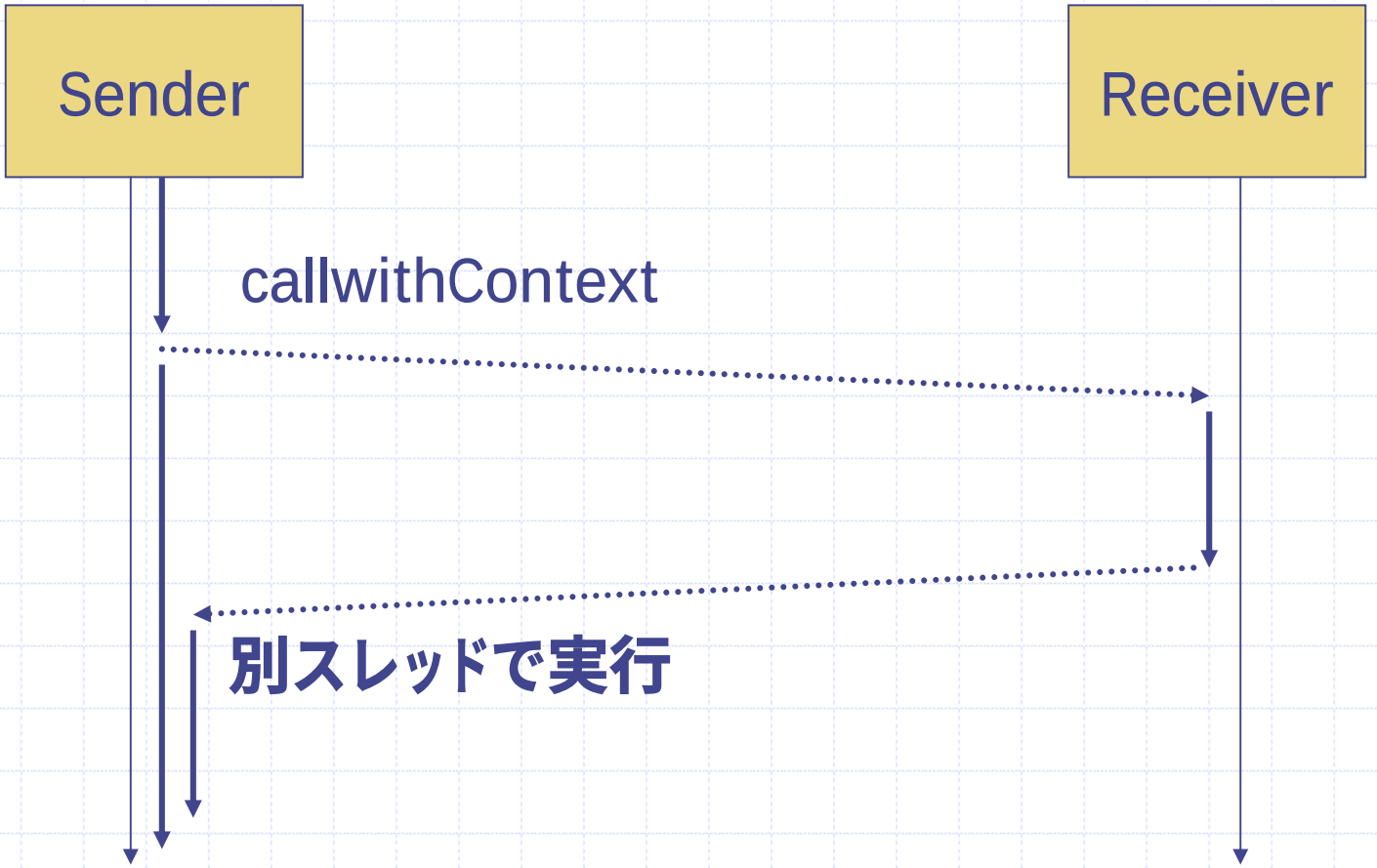
通信モード(2) ブロッキング呼び出し



通信モード(3) Future呼び出し



通信モード(4) Context付き呼び出し

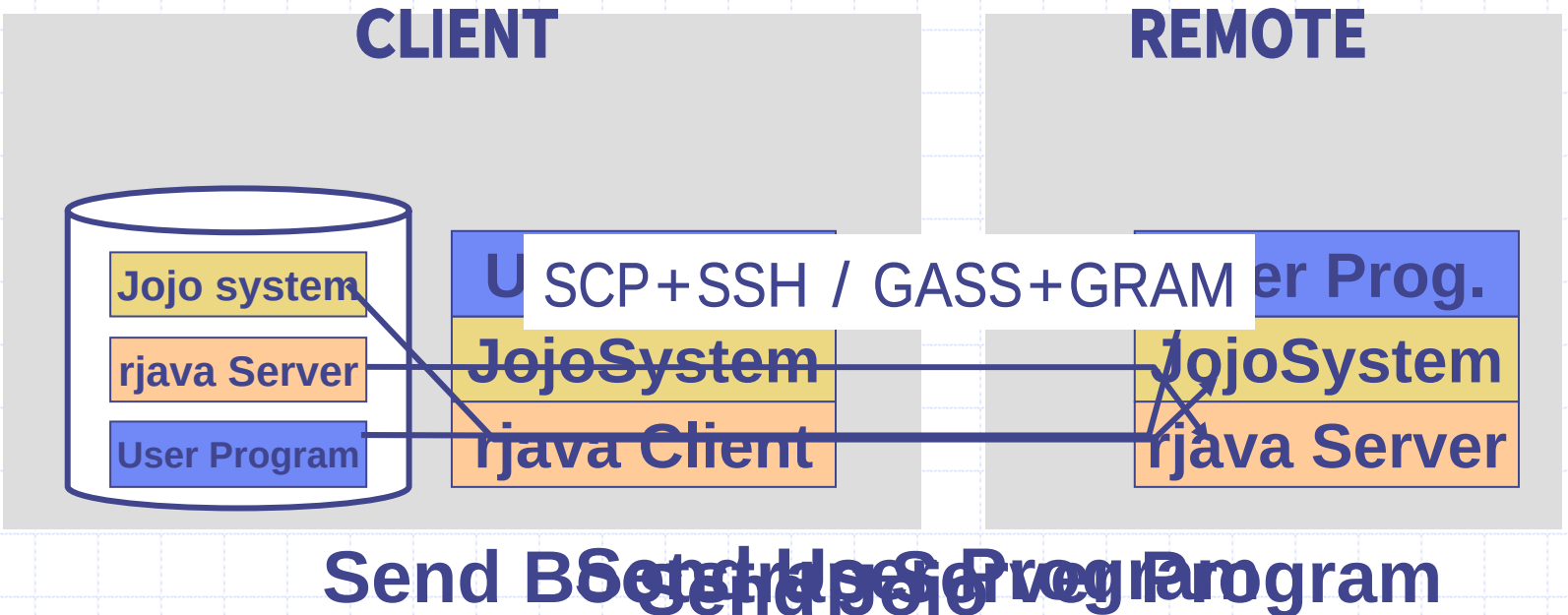


Jojoの起動

- クライアントから設定ファイルを指定して起動
  - プログラムへの引数はプロパティファイルで与える
- 起動にはGlobus,ssh,rshが使用可能
- すべてのプログラムが動的にダウンロードされる
  - ブートストラップサーバrjavaを使用

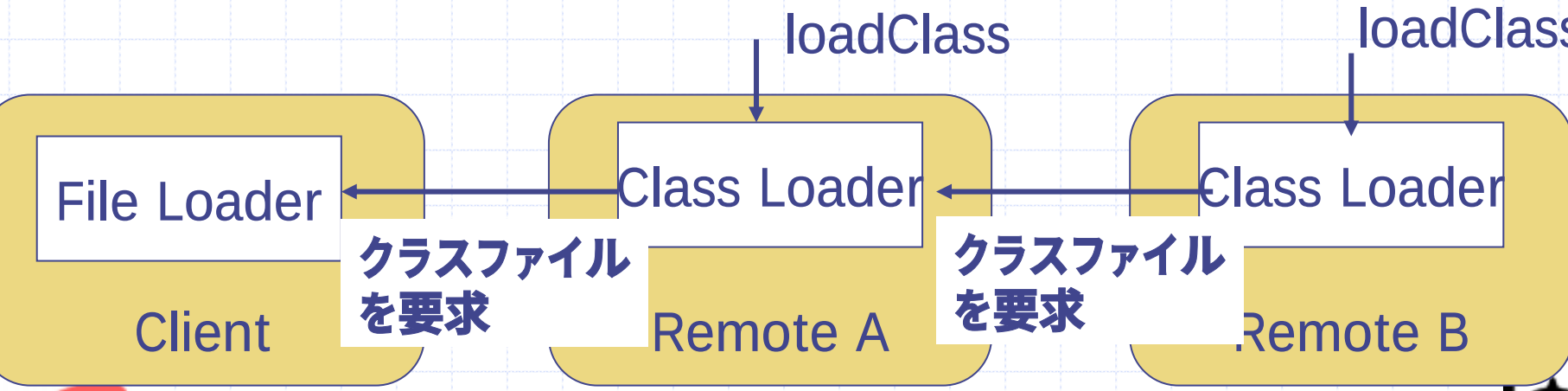
rjavaによるブートストラップ起動

- 段階的にシステムとユーザプログラムを自動転送



クラスファイルの動的ロード

- ブートストラップサーバがクラスローダを提供
  - リモートノード上ではすべてのクラスがこのローダでロードされる
  - クライアントノード上のファイルローダと通信してクラスファイルを取得
  - 2段目以降も同様に
  - ファイルローダはCLASSPATHを参照してクラスファイルをロード。Jar ファイルにも対応



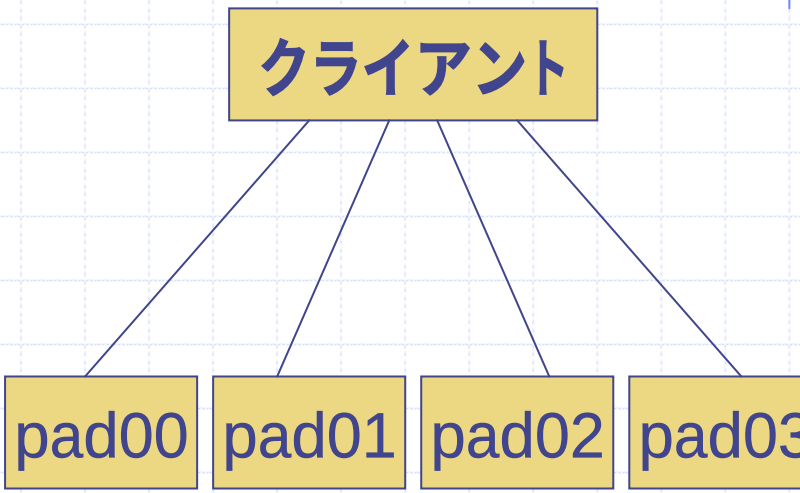
コンフィギュレーションファイル

- XMLで記述
- 階層構造をそのまま表現
- デフォルト値の設定で表記の長さを短縮

```
<!ELEMENT node (code?,invocation?,node*)>
<!ATTLIST node host CDATA #REQUIRED>
<!ELEMENT code (#PCDATA)>
<!ELEMENT invocation EMPTY>
<!ATTLIST invocation
        javaPath CDATA #IMPLIED
        rjavaProtocol CDATA #IMPLIED
        rjavaRsh CDATA #IMPLIED
        rjavaRcp CDATA #IMPLIED
        xtermDisplay CDATA #IMPLIED
        xtermPath CDATA #IMPLIED
>
```

コンフィギュレーションファイル例

```
<node host="root">
 <code> PiMaster </code>
 <node host="default">
  <code> PiWorker </code>
  <invocation
   javaPath="java"
   rjavaJarPath="/tmp/rjava.jar"
   rjavaProtocol="ssh"
   rjavaRsh="ssh"
   rjavaRcp="scp" />
 </node>
 <node host="pad00" />
 <node host="pad01" />
 <node host="pad02" />
 <node host="pad03" />
</node>
```



プログラムへの入力

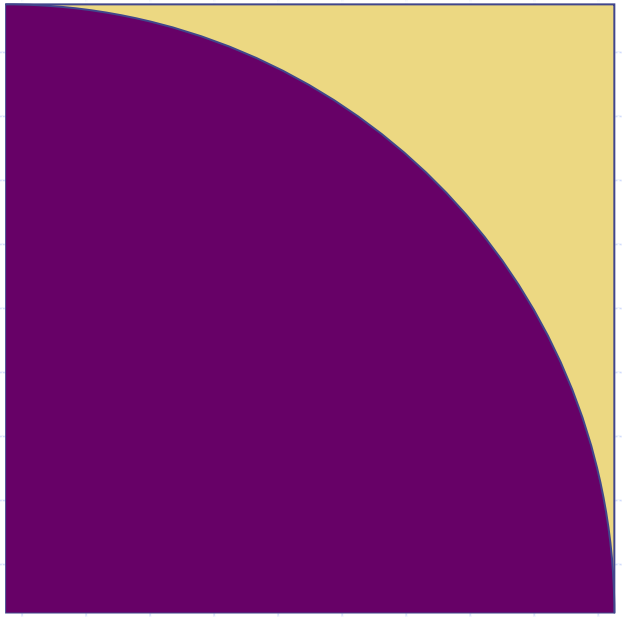
- 起動時にプロパティファイルを指定
  - プロパティがすべてのCodeのinitメソッドに渡される

```
> java silf.jojo.Jojo CONF_FILE  PROP_FILE
```

- 内部的にはMapを渡している
  - 文字列に限らず任意のシリアライズ可能なオブジェクトを渡すことができる

サンプルプログラム

- 乱数で円周率を求める
  - 正方形の中にランダムに点を打ち、円弧に入る確率からΠを逆算
  - マスタ･ワーカ
  - 動的負荷分散
    - ワーカがマスタにアクセスしてタスクを取得



times=100000

divide=10

サンプルプログラム(マスタ)

```
public class PiMaster2 extends Code{
  略
  synchronized public Object handle(Message msg) throws JojoException{
    if (msg.tag == PiWorker.MSG_TRIAL_REQUEST){
      long [] pair = (long[])(msg.contents);
      doneTrial  += pair[0];
      doneResult += pair[1];
      if (doneTrial >= times){
        synchronized (this) {done = true; notifyAll();}
        return new Long(0);
      } else
        return new Long(perNode);
    } else
      throw new JojoException("cannot handle the message: " + msg);
  }
}
```

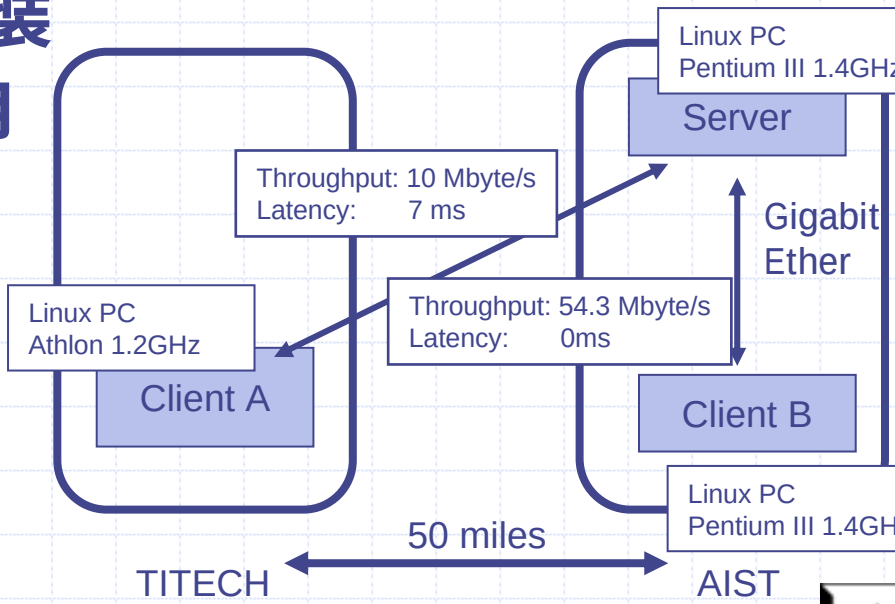
サンプルプログラム(ワーカ)

```
public class PiWorker2 extends Code{
  public static final int MSG_TRIAL_REQUEST = 1;
  Random random = new Random();
  public void start() throws JojoException{
    long trialTimes = 0, doneTimes  = 0;
    while (true){
      Message msg =
         new Message(MSG_TRIAL_REQUEST,
                  new long[]{trialTimes, doneTimes});
      trialTimes =
            ((Long)(parent.call(msg))).longValue();
      if (trialTimes == 0) break;
      doneTimes = trial(trialTimes);
    }
  }
```

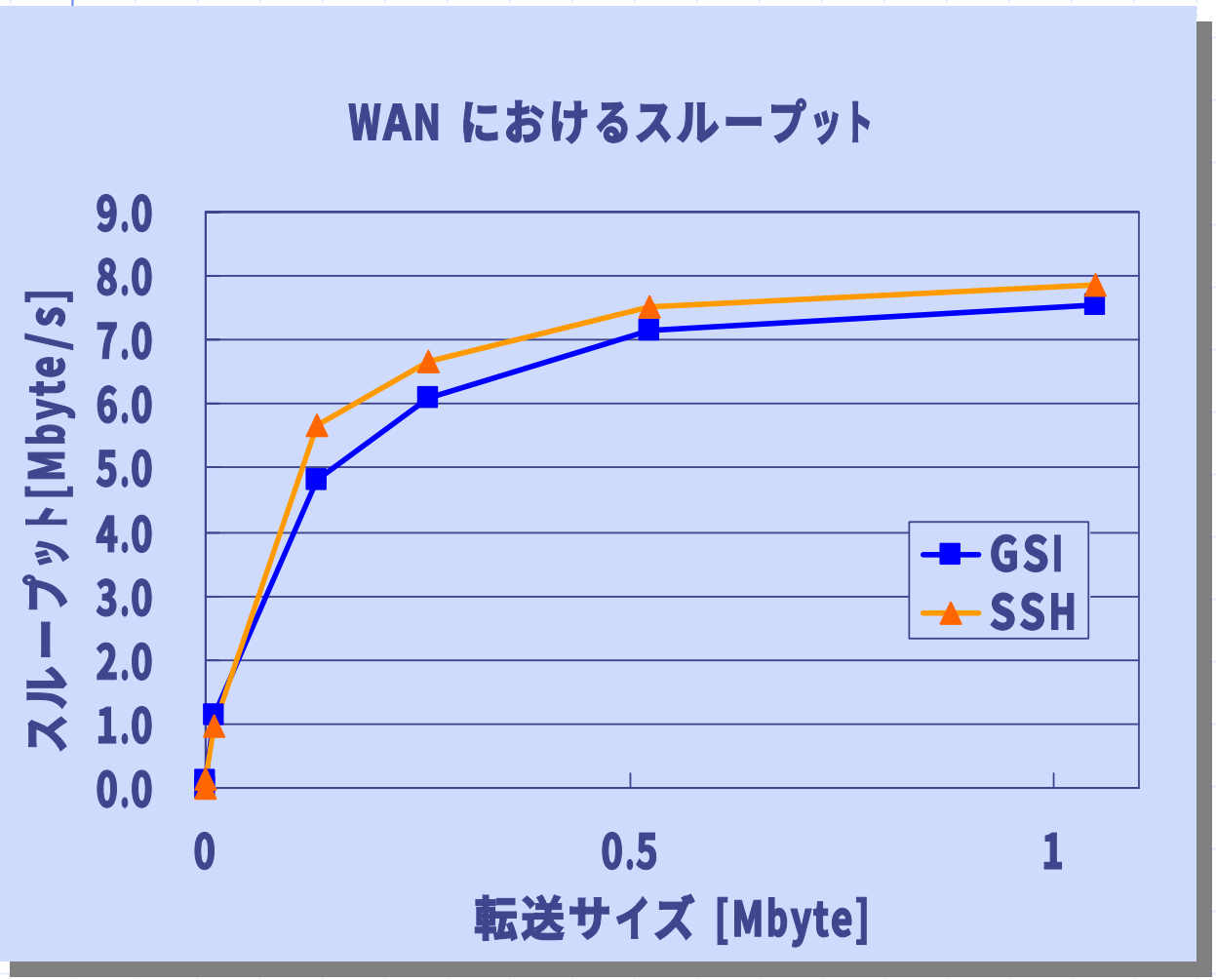
```
  private long trial(long trialTimes){
      long counter = 0;
      for (long i = 0; i < trialTimes;
       i++){
        double x =
       random.nextDouble();
        double y =
       random.nextDouble();
        if (x * x + y * y < 1.0)
       counter++;
      }
      return counter;
    }
}
```

予備実験

- LAN環境とWAN環境でスループットを計測
  - 産総研と東工大
- それぞれ、GSIを使用したものとSSHを使用したもので比較
  - GSIのSSLはPure Java実装
  - SSHは外部コマンドを使用

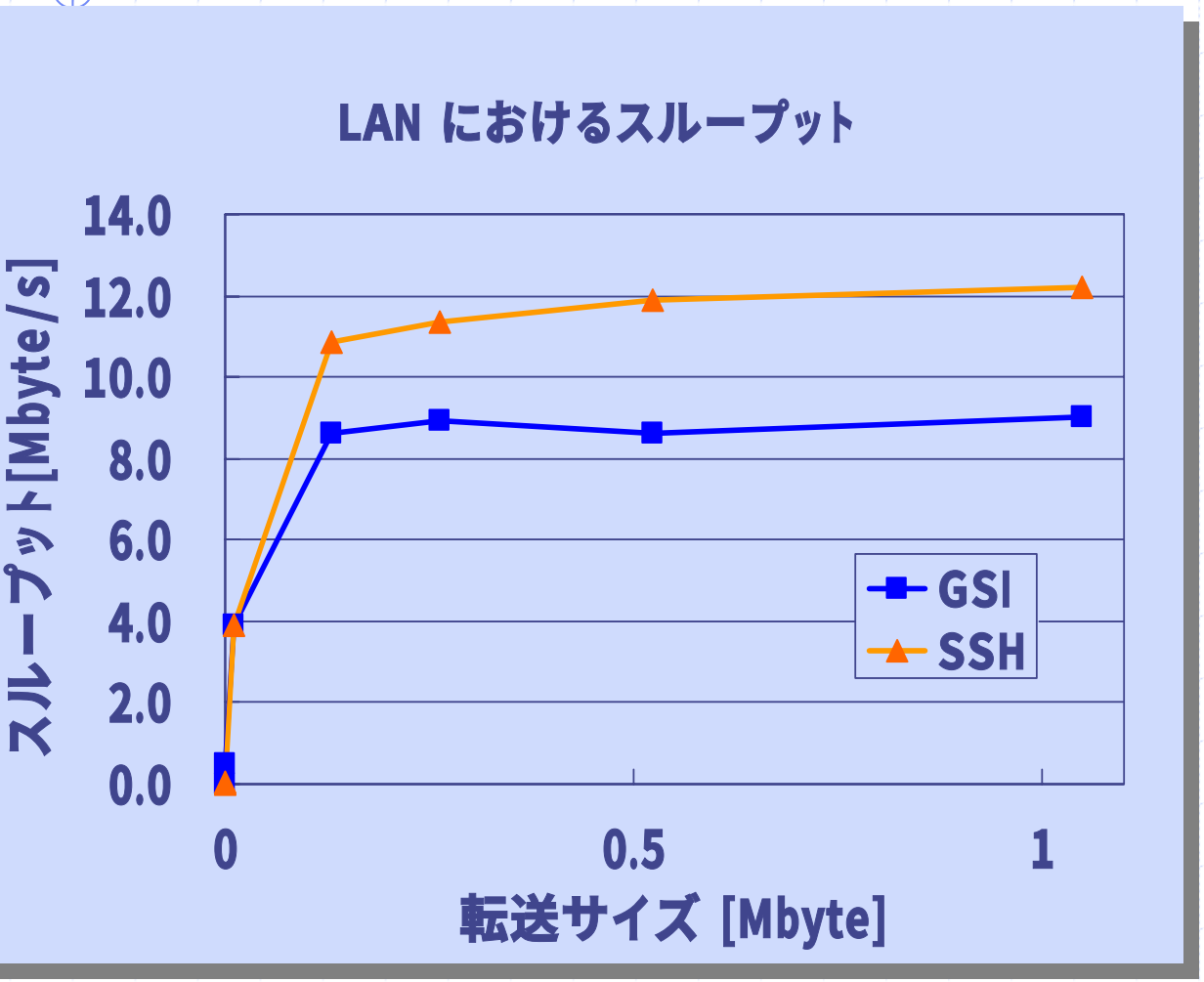


結果(WAN)



- **本来のスループットは**10Mbyte/s
- 7-8**割程度の性能**
- SSH**のほうが高速**

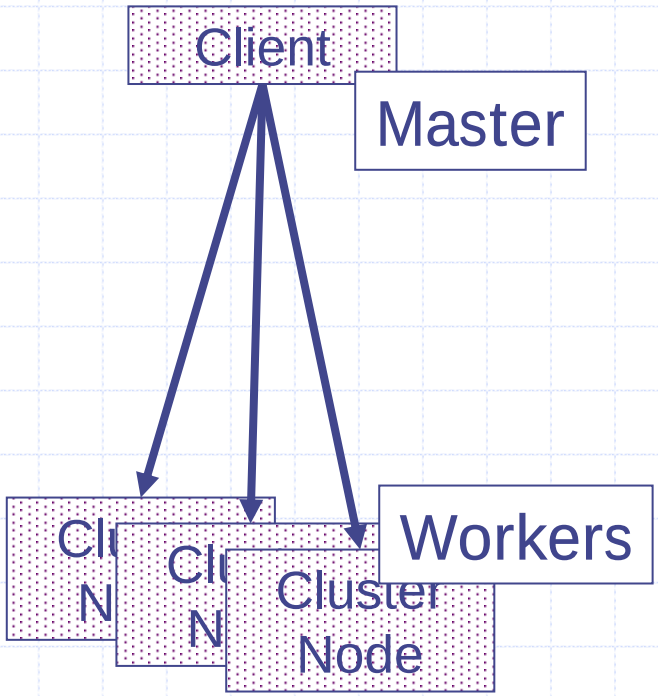
結果(LAN)



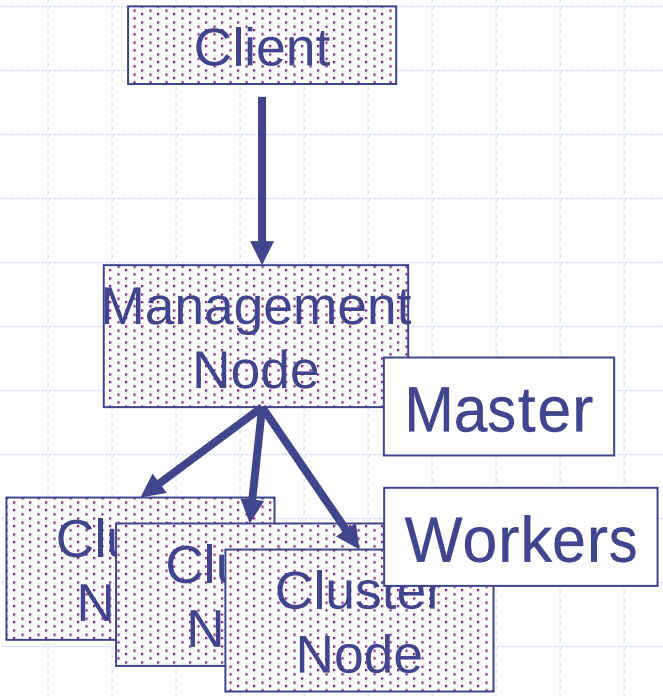
- 本来のスループットは5 4Mbyte/s
- GSIの性能が低い(sshの2/3)
- 双方とも本来のスループットの1/5程度

マスタ･ワーカ 計算による評価

- 2層モデルと3層モデルで性能を比較



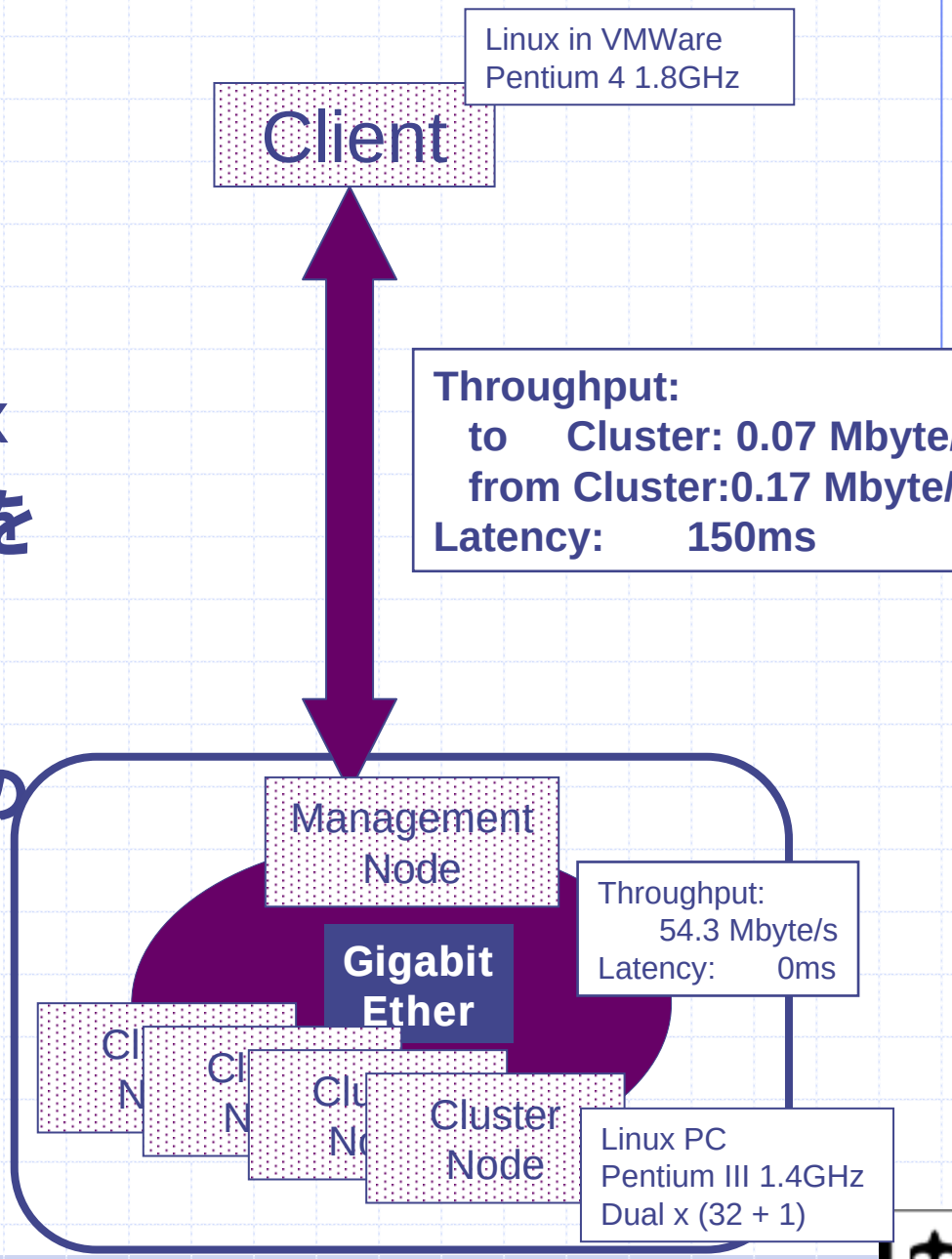
2layers model



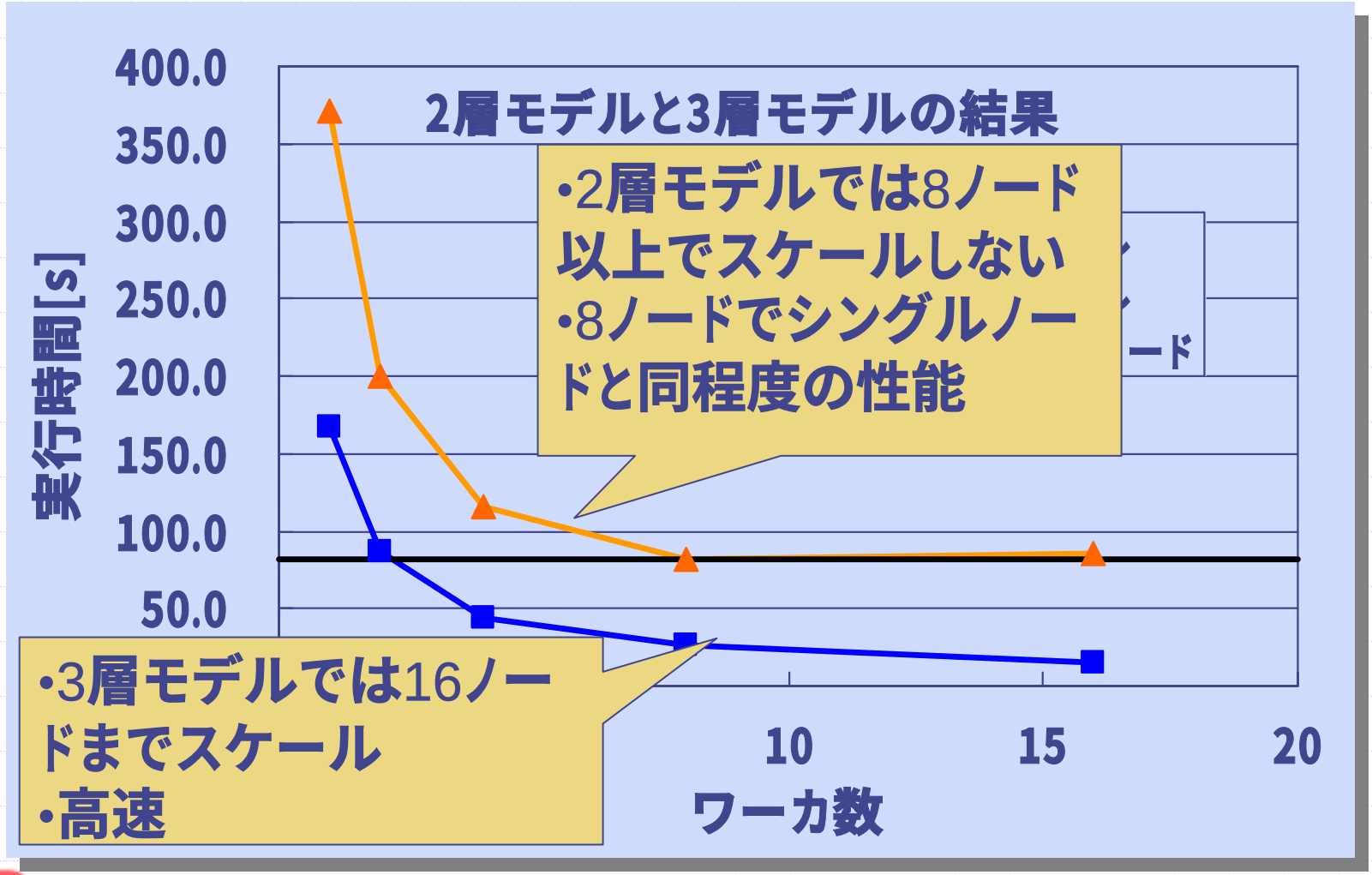
3layers model

評価環境

- CATV＋無線LAN
- VMWareのなかのLinux
- 円周率計算プログラムを使用
  - 10^8の試行を10^4に分割
  - 1つのタスクのワーカでの実行時間は8ms程度



マスタ･ワーカ計算の結果



マスタ･ワーカ実験の考察

- データ転送量は数バイト
  - データ転送時間は無視できる程度だが、レイテンシは無視できない
- ワーカでの実行時間は8ms程度
  - マスタ･ワーカ実行には不利な条件だが3層モデルでは効率的に実行可能

まとめ

- 階層的なグリッド環境に適した実行環境Jojoの設計と実装を述べた
- 簡単なサンプルプログラムを示した
- 予備的性能評価を行いその結果を示した

今後の課題

- 最適化問題システムであるjPoPにJojoを適用し Jojoのスケーラビリティを確認する
  - GAによるNMR画像からのたんぱく質構造決定
- システム的な改良を検討する
  - Sibling間通信の直接通信化
  - ノードに対する参照の転送