オーバレイスケジューラJojo3のGridRPCへの適用

産業技術総合研究所 情報技術研究部門

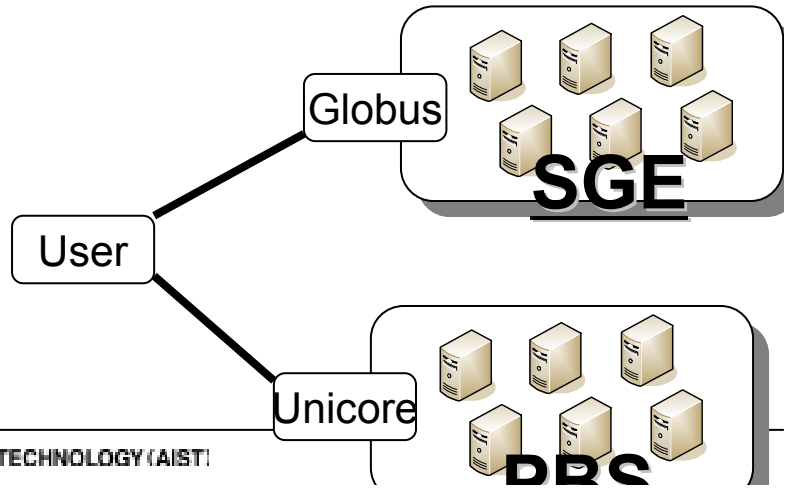
中田秀基, 田中良夫, 関口智嗣

背景

- グリッド技術とクラスタ技術の発展
  - 個々のユーザがアクセス可能な資源量の増加
- にもかかわらず有効に活用できていない
  - 有効利用できるのは簡単な独立ジョブのみ
  - 問題：分散プログラミングが困難
    - 分散プログラミングの本質的な困難性
    - ジョブ起動，通信の確立の困難性
      - 環境の不均質性，非対称性

背景(2)

- アプリケーションレベルスケジューラ
  - 特定のアプリケーションパターンに特化したフレームワーク
  - 下位レイヤを隠蔽．容易にプログラミングが可能
  - Ex. GridRPC, Condor MW，jPoP
- アプリケーションレベルスケジューラの実装を補助するオーバレイスケジューラを提案



背景(3)

- Jojo3[08' HOKKE] を提案
  - 有効性が十分に確認されていない
    - 十分なアプリケーションスケジューラが実装できていない.
  - クライアントプロトコル
    - 多言語に対応するように設計されているが, Javaのみでしか実装されていないので確認できていない.

研究の目的

- オーバレイスケジューラJojo3の有効性を確認
  - 複雑なアプリケーションスケジューラであるNinf-GをJojo3を用いて実装
  - 実装コストを行数で評価
- Jojo3のクライアント インターフェイスが容易に実装できることを確認
  - Pythonでスレッドを使用せずに実装

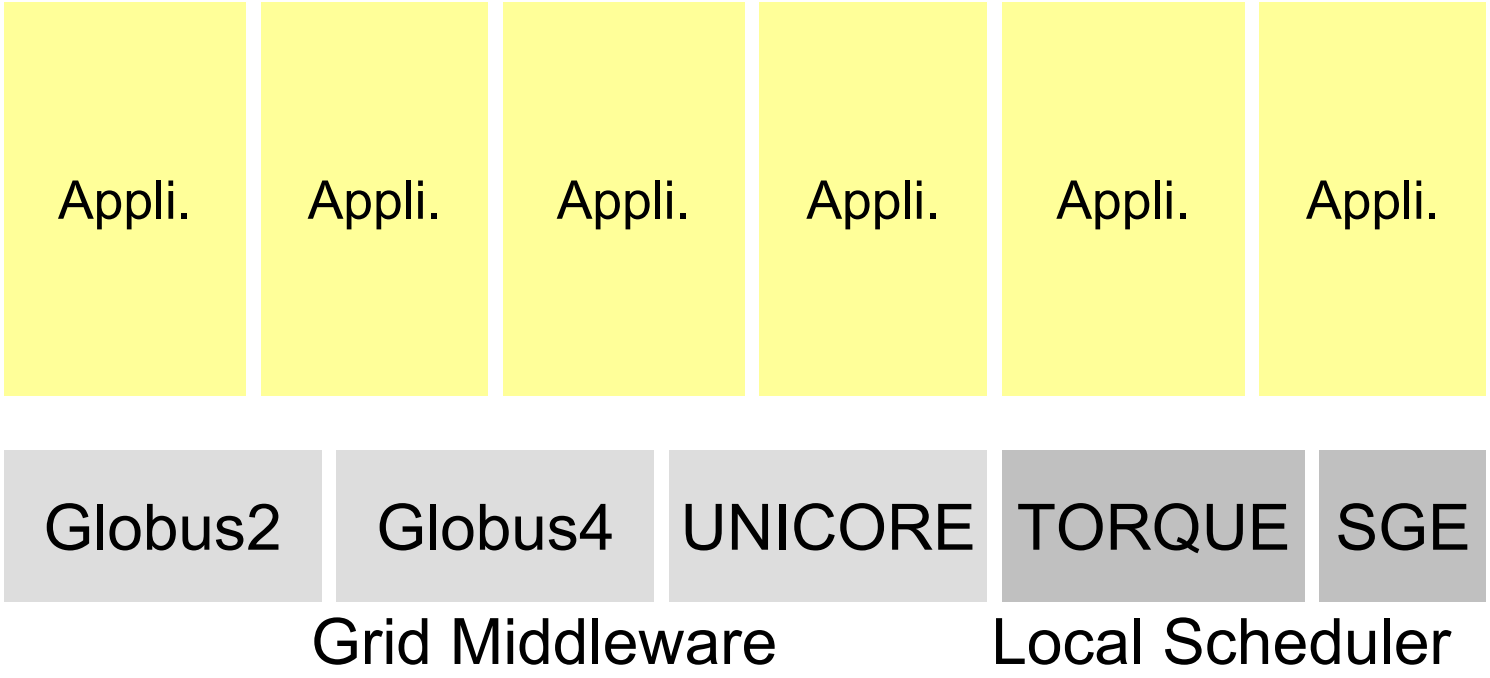
アウトライン

- オーバレイスケジューラ Jojo3の概要
- GridRPC Ninf-G5の概要

- Jojo3によるNinf-G5の実装
- PythonによるJojo3クライアントの実装

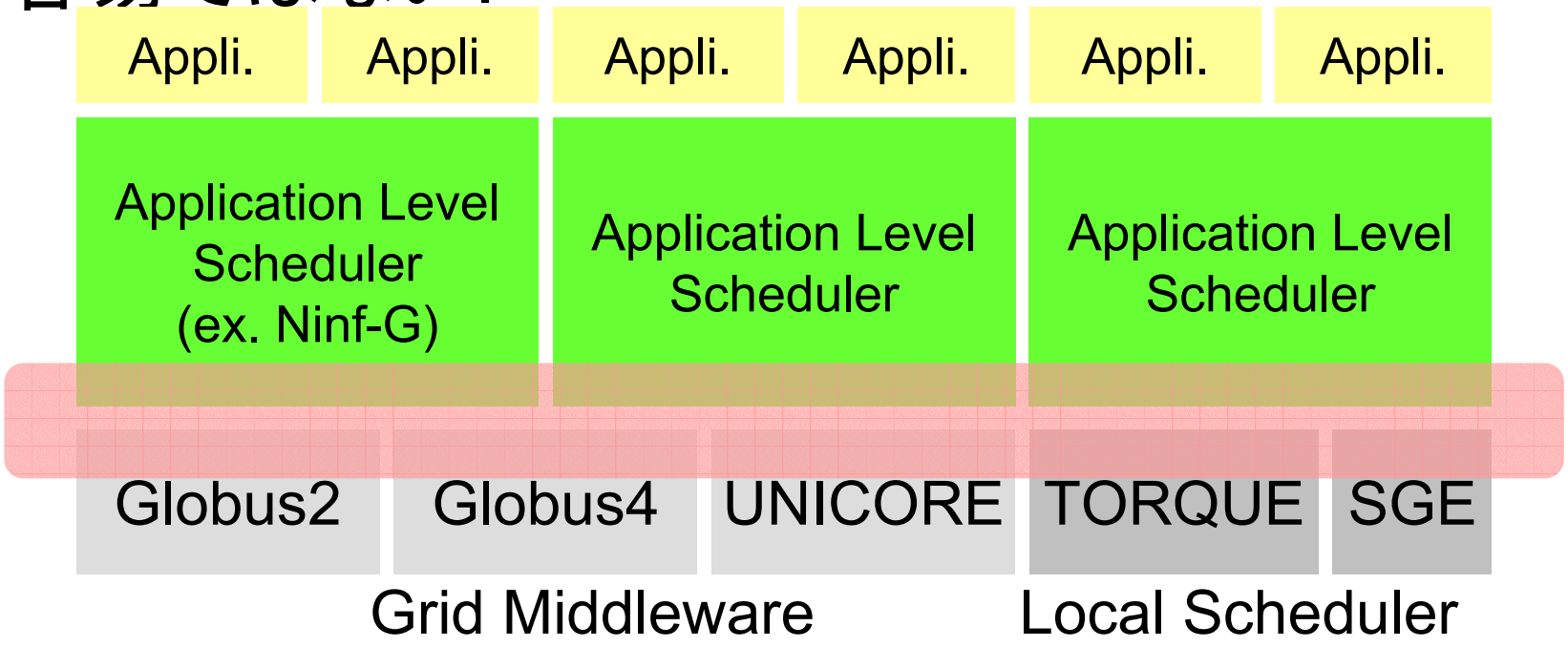
なぜグリッド上のプログラミングは面倒なのか

- さまざまなグリッドミドルウェア
  - 資源のモニタリング，ジョブ起動の作法が違う
- ジョブが起動するまでにどれだけ時間がかかるかわからない
  - バックエンドのキューイングシステムの状態に依存
- ジョブ間のネットワーク接続が提供されない
  - ネットワークの非対称性をプログラマが解決しなければならない.



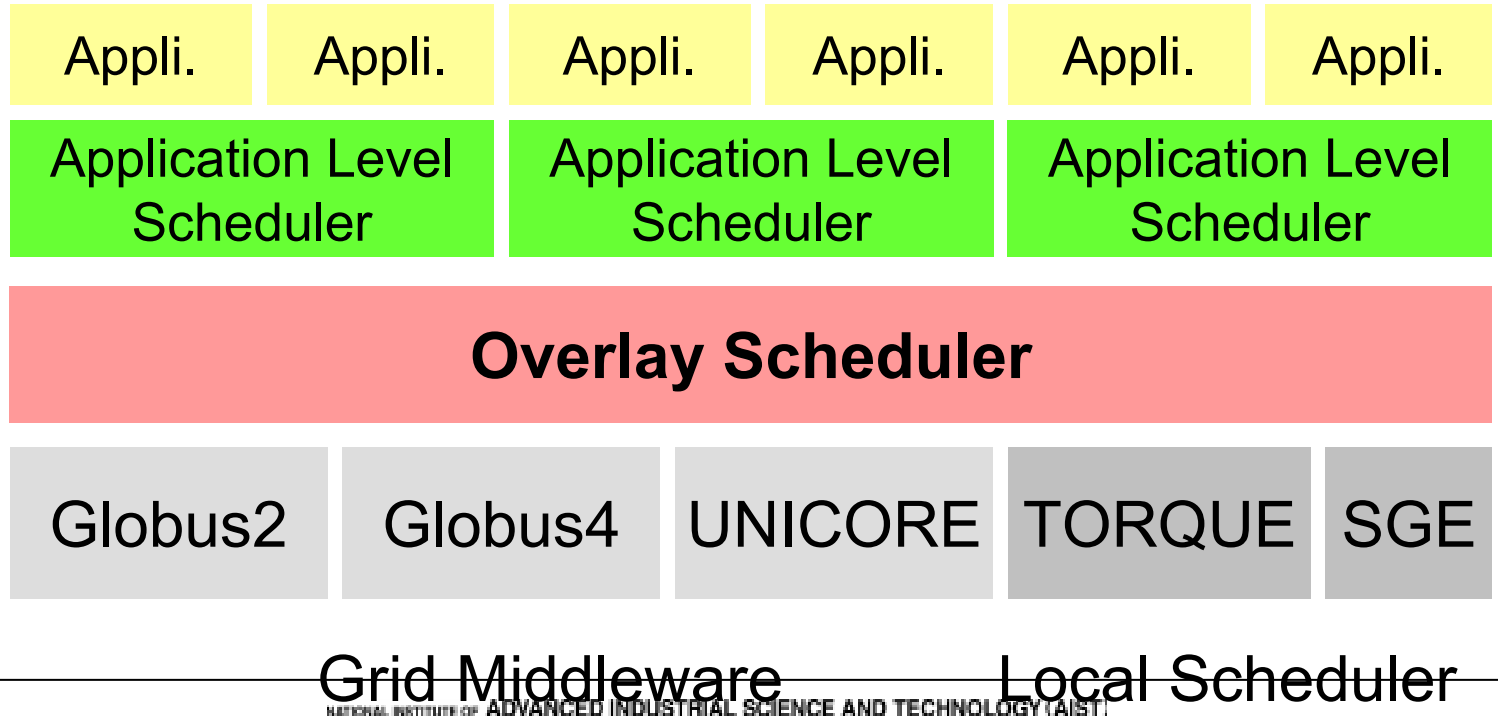
アプリケーションレベルスケジューラ

- グリッドの複雑さを隠蔽してアプリケーションプログラミングを容易に
  - Ex. Ninf-G5, Condor-MW
- アプリケーションレベルスケジューラの実装は容易ではない.



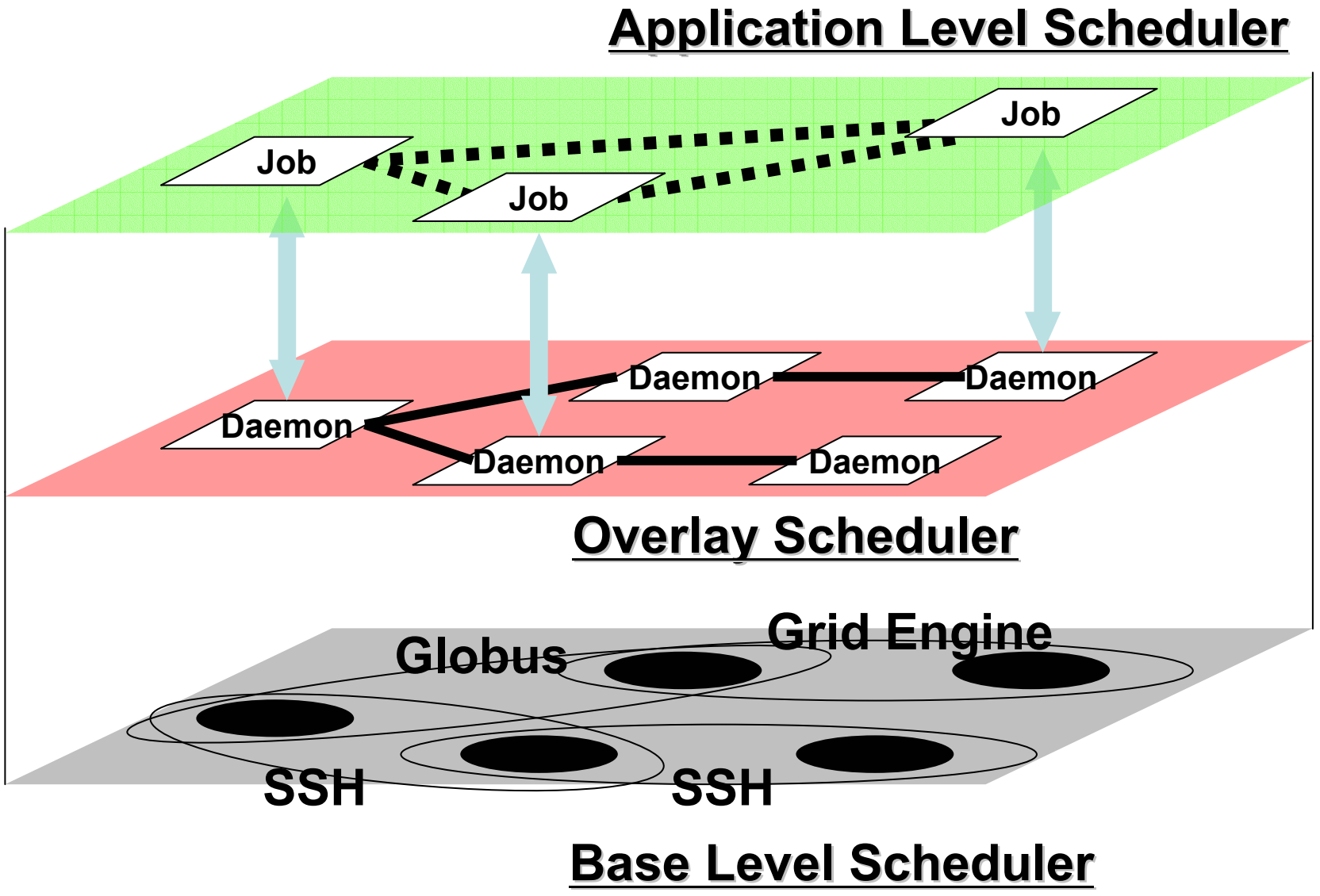
オーバレイスケジューラの導入

- アプリケーションレベルスケジューラと，グリッドミドルウェア，キューイングシステムの間に，新たな層を導入
- アプリケーションレベルスケジューラ実装を容易に



オーバレイスケジューラの機能

- 資源をホモジニアスに提供
  - グリッドミドルウェア，キューイングシステムの実装によらず一元的に管理
  - 各計算資源のモニタリング情報を提供
- ジョブの起動
  - どの計算資源でも同一のインターフェイスで即座に起動
- ジョブ間通信
  - 任意のジョブの間で通信を提供

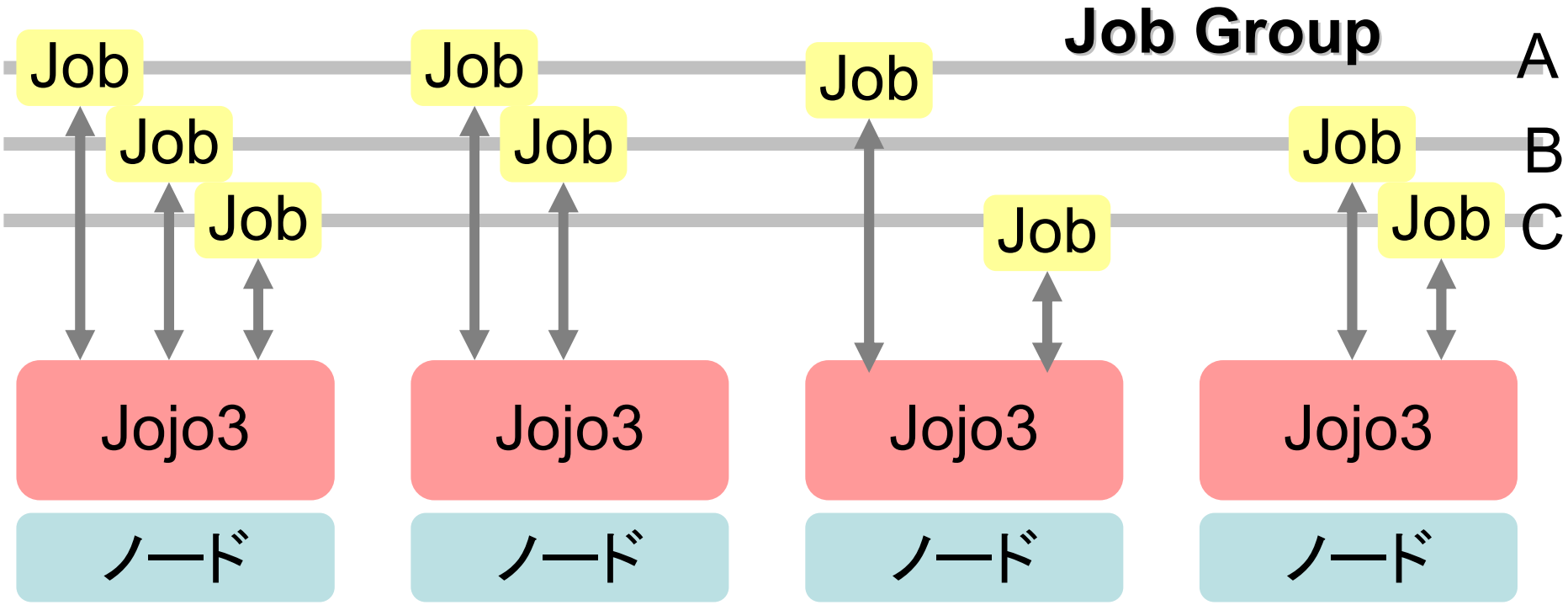

Application Level Scheduler
Job
Job
Job
Daemon
Daemon
Daemon
Daemon
Daemon
Overlay Scheduler
Globus
Grid Engine
SSH
SSH
Base Level Scheduler

Jojo3の設計

- 機能
  - 使用可能なノード群の管理
  - ジョブの起動と管理
  - ジョブ間通信の提供
- アーキテクチャ
  - ルートとなるノードから再帰的にデーモンを起動してデーモンのネットワークを構成
  - デーモンはジョブに対してインターフェイスを提供

Jojo3の構成

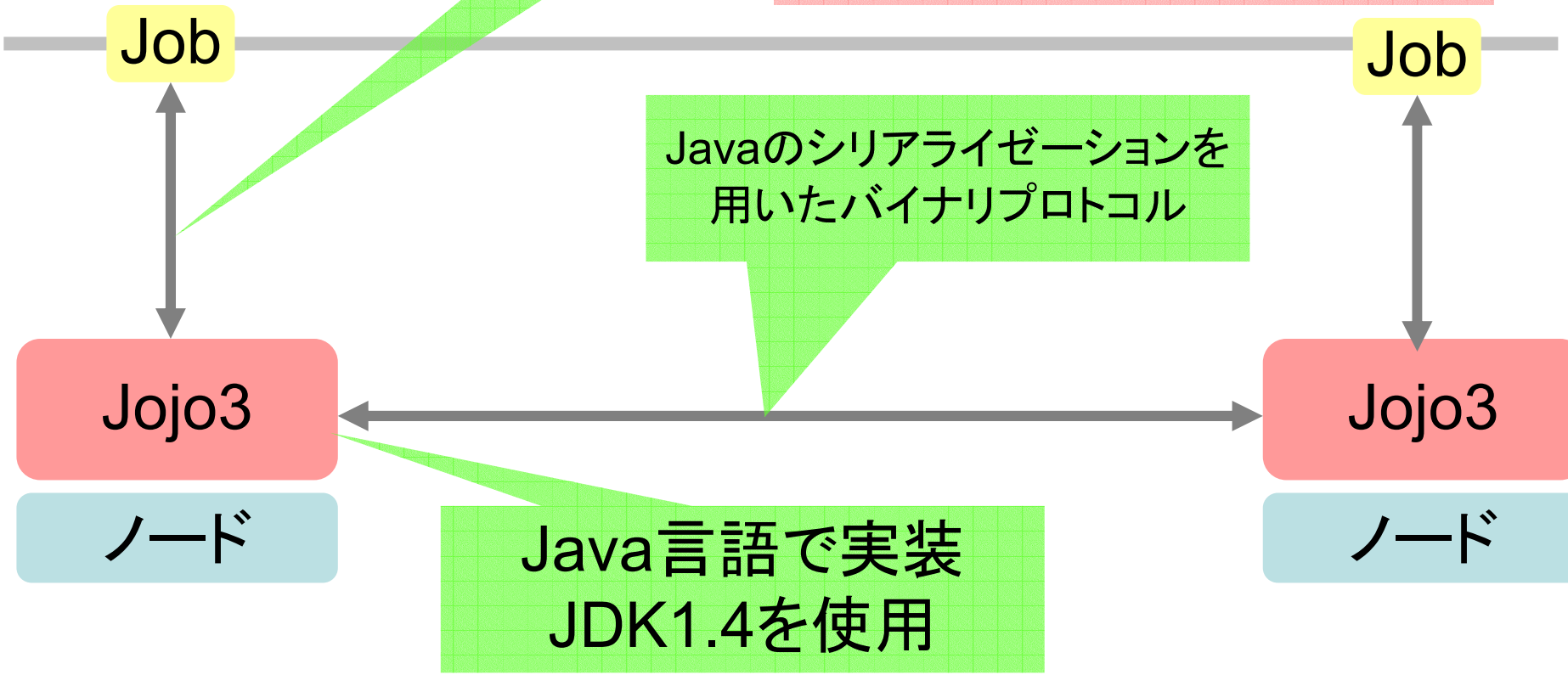
- Jobグループ – ルートジョブから起動されたジョブで構成
- 複数のJobグループで基盤を共有



実装とプロトコル

JSONを用いた
テキストベース
プロトコル

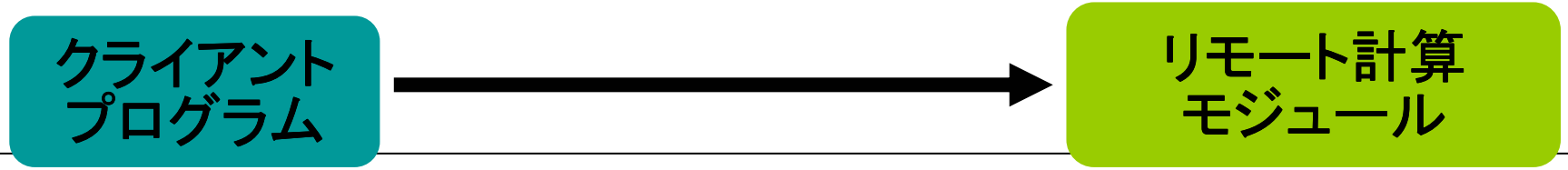
- 言語，アーキテクチャ非依存
- オブジェクトとのマッピングが容易
  - XMLと比較して，恣意性がなく，
    スキーマを記述する必要がない



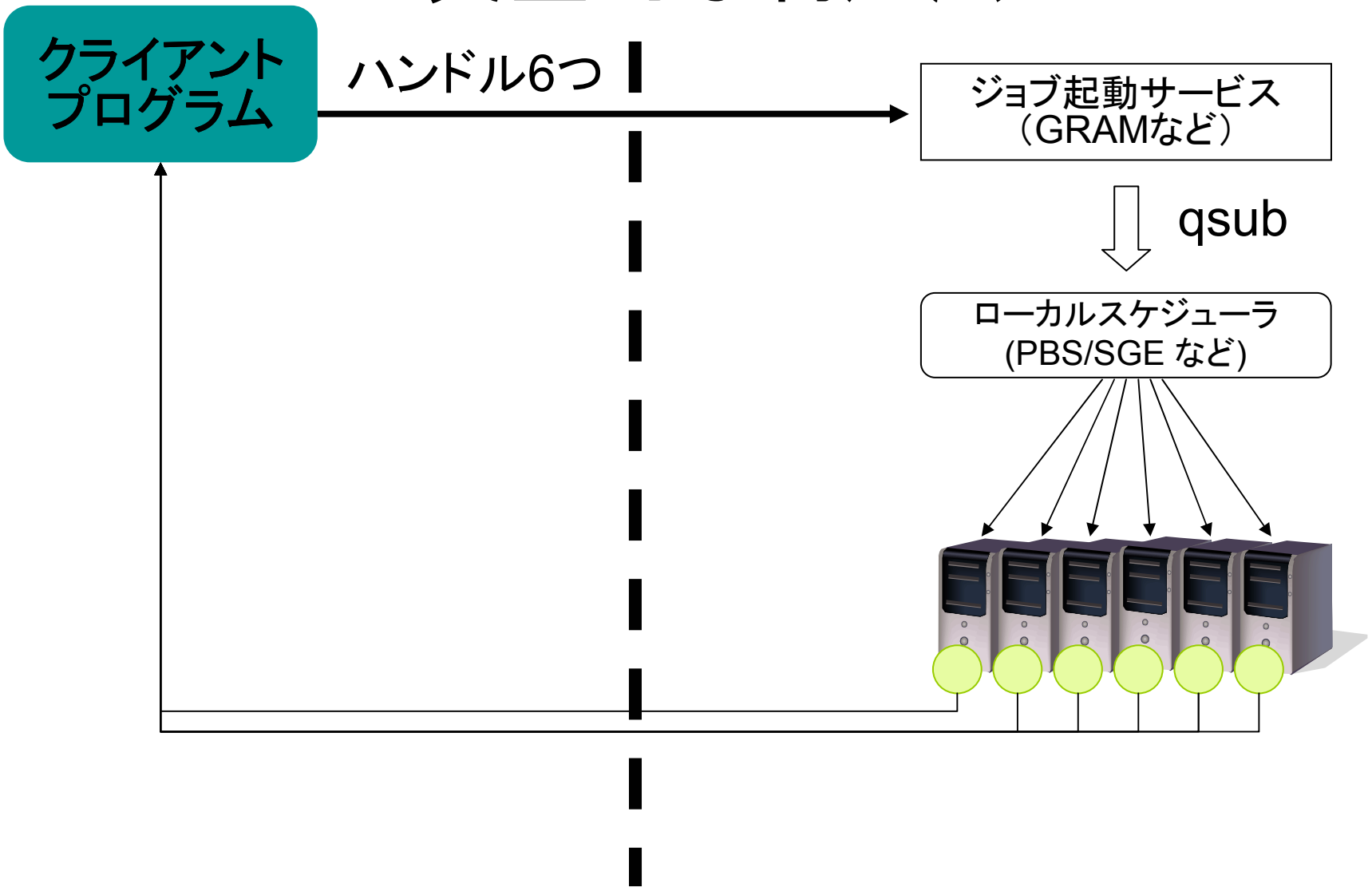
Ninf-G

- グリッド上でRemote Procedure Call を実現
  - サーバ側の関数をクライアント側から使用
  - 引数転送は暗黙裡に行われる

```
/* initialize function handle */
  for (i = 0; i < NUM_HOSTS; i++)
    grpc_function_handle_init(&handles[i], hosts[i], port, "pi/pi_trial");
  for (i = 0; i < NUM_HOSTS; i++)
   /* parallel invocation */
    gprc_call_async(&handles[i], i, times, &count[i]);
  /* wait for all the calls */
 grpc_wait_all();
```

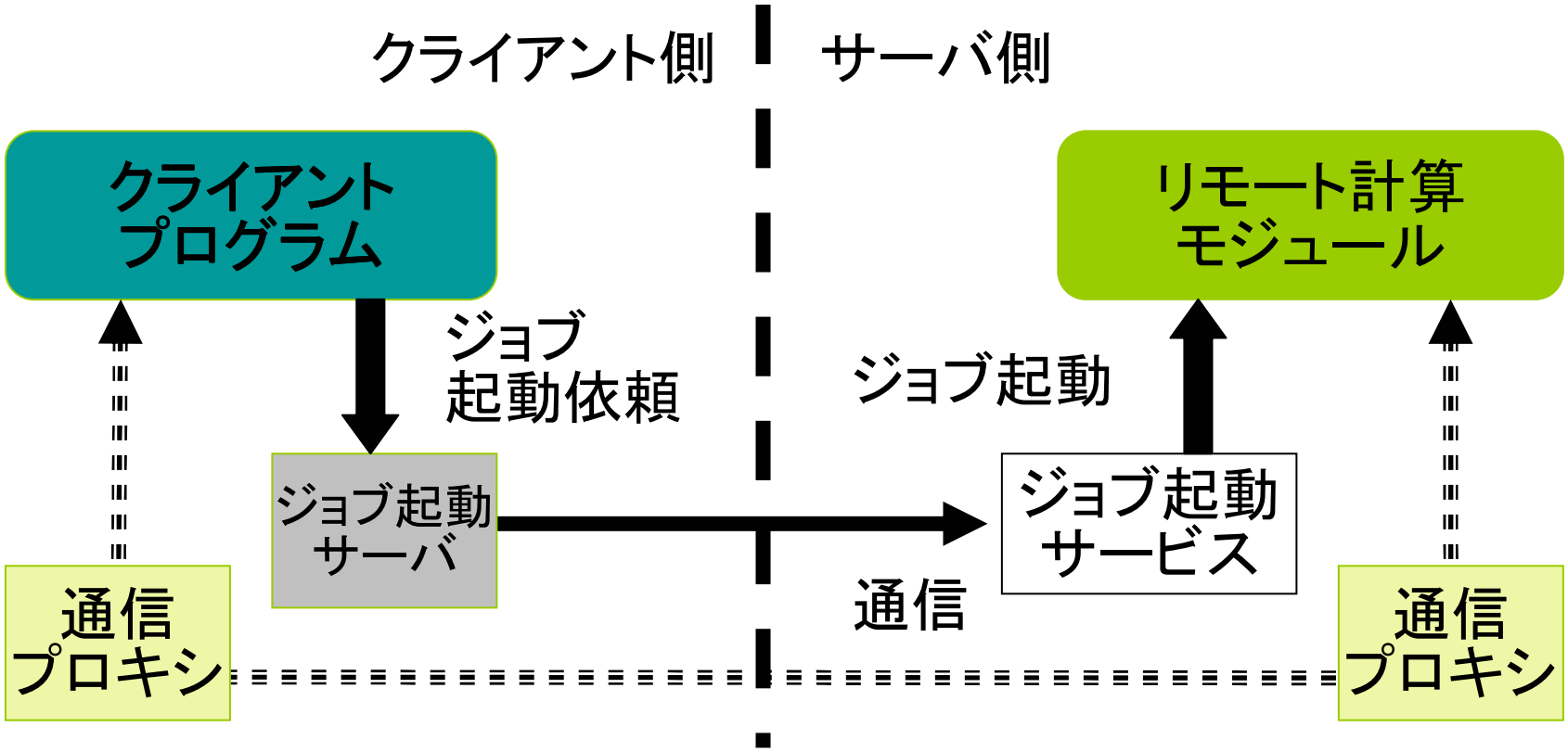


典型的な利用法

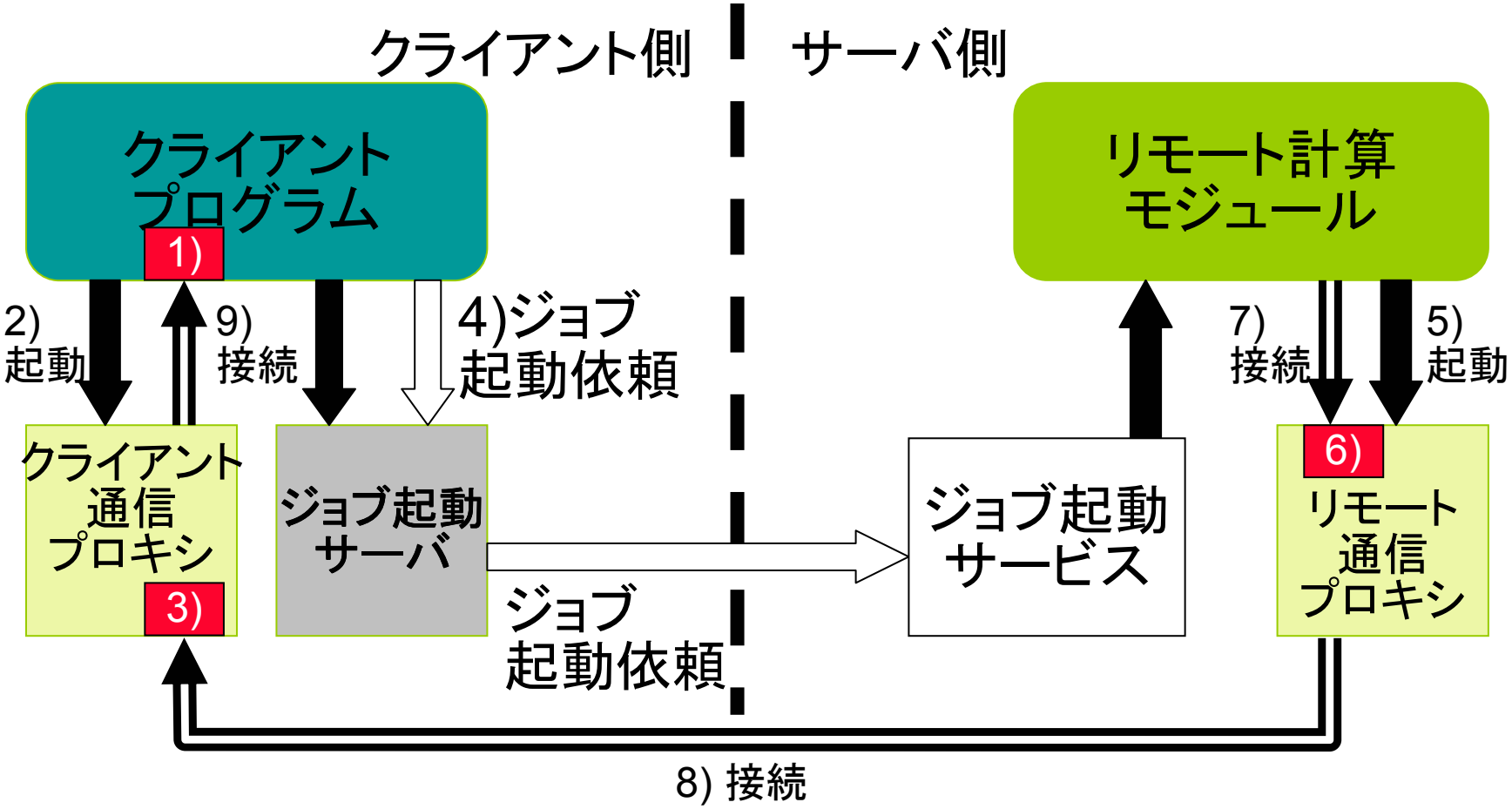

クライアント
プログラム
ハンドル6つ
ジョブ起動サービス
（GRAMなど）
qsub
ローカルスケジューラ
(PBS/SGE など)

ジョブ起動サーバと通信プロキシ

- 個々のグリッドミドルウェアへ依存するコードをライブラリのコア部分から排除

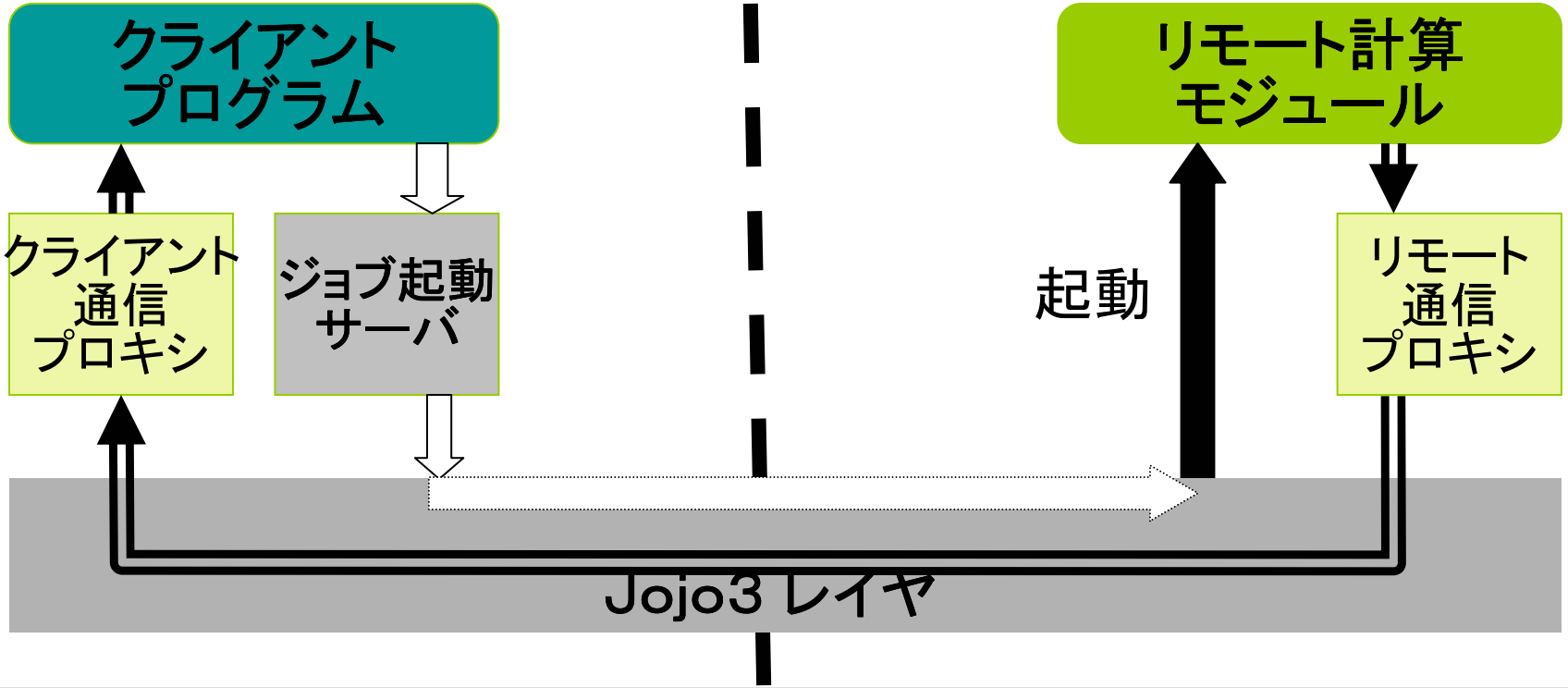


起動と通信接続のシーケンス



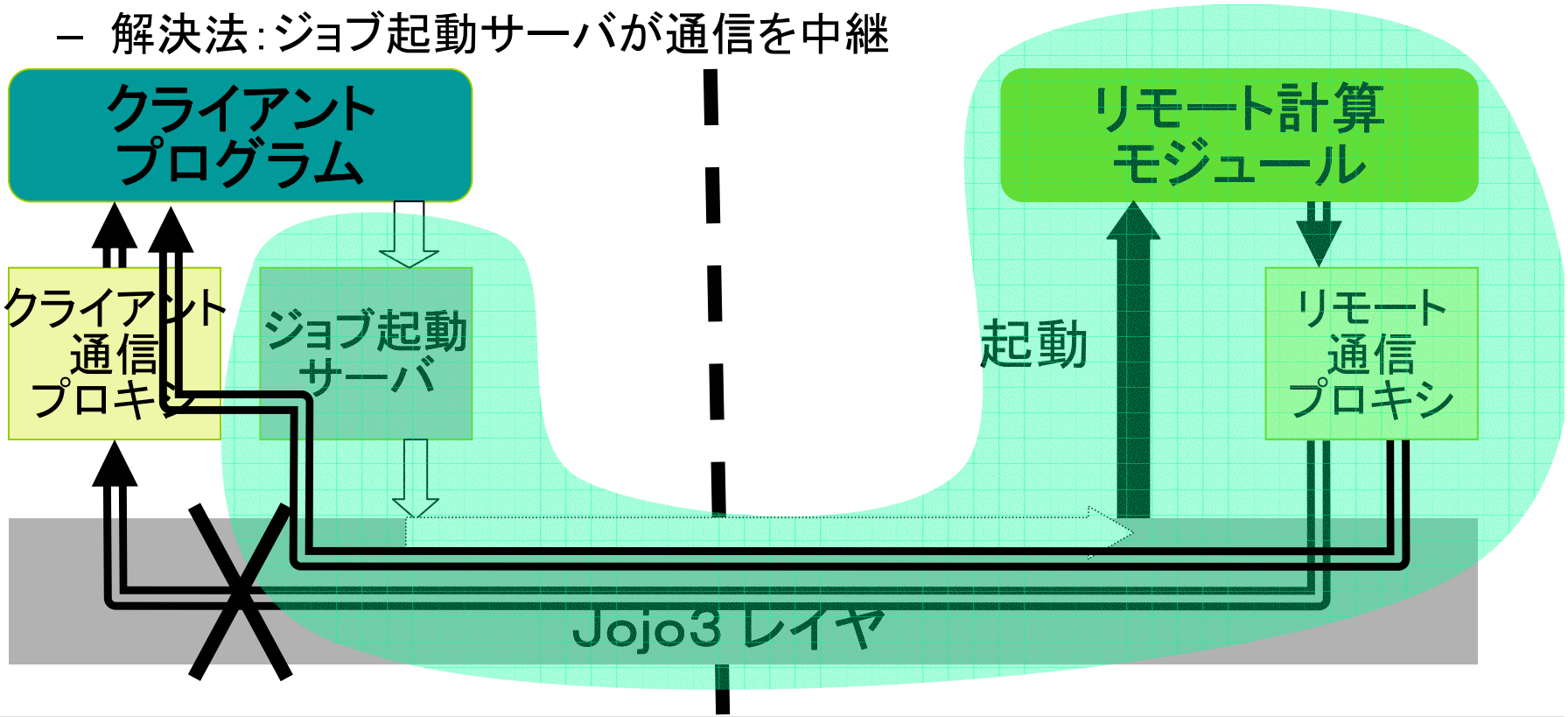
Jojo3による実装

- ジョブ起動サーバと，通信プロキシをJojo3で実装すればよい

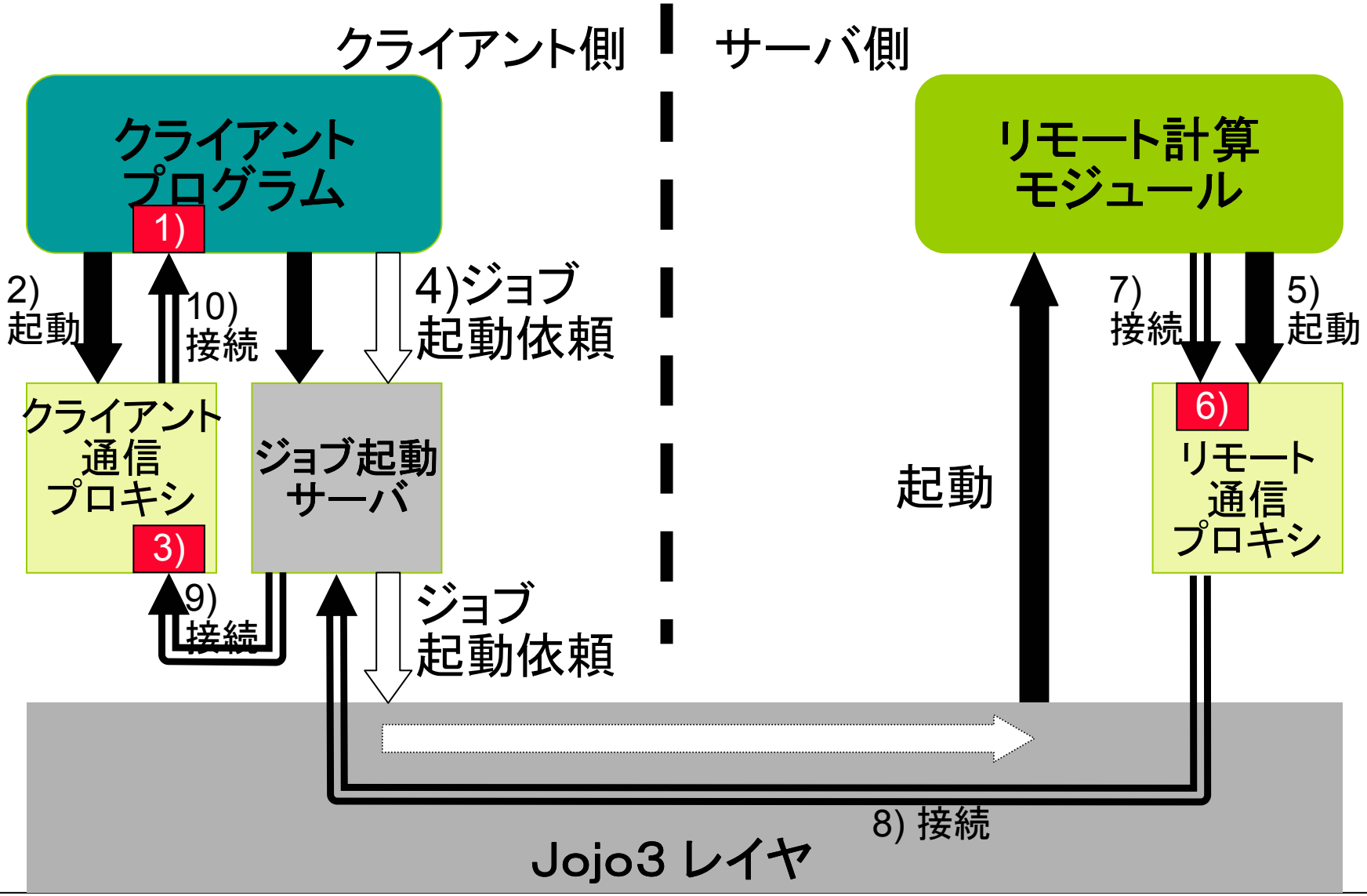


Jojo3による実装 －問題点

- ジョブ起動サーバと通信プロキシは独立したプロセスで，それぞれクライアントから起動される
  - クライアント通信プロキシとリモート通信プロキシが同じジョブグループに入れない - 通信できない
  - 解決法：ジョブ起動サーバが通信を中継



Jojo3によるNinf-Gの実装



ジョブ起動サーバの記述行数

- Javaで実装
  - Condor/NAREGI-MW向けと一部コードを共有
    - クライアントとの通信プロトコル処理など

比較的少量の
コードで記述

	固有部	共有部	計
Jojo3	315	1176	1491
Condor	340		1416
NAREGI-MW	1535		2711

通信プロキシの記述行数

- 通信プロキシはPythonで実装
  - ソケット通信版をベースに　　変更は33行

比較的少量のコードで記述

	固有部	共有部	計
Jojo3 -リモート	74		697
ソケット - リモート	51	623	674
ソケット - クライアント	52		675

PythonクライアントAPIの設計と実装

- 基本的にJavaのAPIのコンセプトを踏襲
  - Non-blocking
  - コールバックベース
- Lambdaによるクロージャを利用できるように修正
  - コールバックは関数で指定
  - c.f. Javaではリスナクラスが一括でハンドル
- スレッドを用いずに実装

Java Client API

```
class Client {
  // コンストラクタ
  Client(UpdateListener listener);

  // 自分のジョブ識別子取得
  String getJobId();

  // 自分の所属するジョブグループの識別子取得
  String getJobGroupId();

  // ジョブグループの起点ジョブの識別子取得
  String getRootJobId();

  // ノード群の情報取得
  NodeInfo [] getNodeInfo();

  // 自分の属するジョブグループの他のジョブの状態取得
  JobState [] getJobState();

  // ジョブの接続要求
  String invoke(String nodeId, JobDescription job);

  // 他のノードへの接続要求
  String connect(String jobId);
}
```

```
interface UpdateListener {
  // 接続成功
  void connected(String conId, Connection con);

  // 接続失敗
  void connectionFailed(String conId);

  // ジョブグループに属するジョブの状態変化
  void jobStateChanged(String jobId,
                       JobState jobState);

  // ノードの状態変化
  void nodeStateChanged(NodeInfo nodeInfo,
                        NodeState nodeState);
}
```

Python Client API

```
class client:
  def __init__(self, updateNodes=None, acceptHandler=None):
    """インスタンス生成メソッド.
     'updataeNodes' はいずれかのノードの状態が変化した際に呼び出されるコールバック関数.
          引数は, nodeInfoオブジェクトのリスト.
    'acceptHander' は, 他のジョブからのコネクションが来た際に呼び出されるコールバック関数.
          引数は, ソケット."""
  def getNodeInfo(self):
    """現在使用可能なノードの情報を取得する. 返り値は nodeInfoオブジェクト. """
  def getJobState(self):
    """ジョブグループ内のジョブの状態を取得. 返り値は jobStatus オブジェクトのリスト."""
  def connect(self, nodeId, jobId, connect_callback = None):
    """他のジョブに対して接続を依頼するメソッド.
    'connect_callback' は 接続した際に呼び出されるコールバック関数.
     これを指定しないとブロック呼び出しになり, 接続が成功(または失敗)
    するまで呼び出しがブロックする. """
  def invoke(self, node, jobDesc, update_callback = None):
    """ジョブ起動メソッド.
    'update_callback' は, 起動したジョブの状態が変化した際に呼び出されるコールバック関数."""
  def selectLoop(self):
    """Jojo3クライアントライブラリに制御を渡すためのメソッド. """
```

記述例

```
## マスタプログラム
class master:
  'master クラスの定義. '

  def addWorkerSocket(self, cid, s):
    'workerからの接続を追加'
    ...
ms = master()  # master オブジェクト作成

# クライアント初期化
cl = jojo3.client.client(
     acceptHandler=ms.addWorkerSocket)

# ノード情報取得
for node in cl.getNodeInfo():
  # ジョブ記述を作成
  jd = jobDescription(args = [...])
  # ノード上でジョブを起動
  cl.invoke(node, jd)

#制御をライブラリに渡す.
cl.selectLoop()
```

```
## ワーカプログラム
class worker:
  'ワーカクラスの定義'
  def __init__(self, s):
    '初期化'

  def start(self):
    'worker 実行'

# client を初期化
cl = jojo3.client.client()
# ルートジョブの取得
rootJob = cl.getJobState()[0]
# ルートジョブに接続
(s, conId) = cl.connect(rootJob.nodeId,
                        rootJob.jobGId.jobId)
# ワーカスタート
worker(s).start()
```

JSONの扱い

- JSON-pyを利用
  - JSON文字列とpythonのmap, arrayを変換
  - Pythonのオブジェクトとの間では変換してくれない.
- JSON-pyをラップする形でオブジェクトとの間の変換ライブラリを実装
  - オブジェクト宣言時にメタ情報をクラス変数として記述する.

```
class jobDescription(object):
    args = [str]          #文字列のリスト
    envs = {str:str}      #文字列のディクショナリ
    workingDirectory = str #文字列
    def __init__(self):
        self.args = []
        self.envs = {}
        self.workingDirectory = None
```

Python クライアントのまとめ

- スレッドなしで実装が可能なことを確認
- プロトコルが多言語で実装可能なことを確認

おわりに

- オーバレイスケジューラJojo3の有効性をGridRPCシステムへの適用で確認
  - Ninf-G側のコーディングは軽微
- Jojo3のクライアントプロトコルの実装容易性を確認
  - Pythonによる実装

今後の課題

- 他の言語でのクライアントライブラリの実装
  - C言語
- さまざまなアプリケーションレベルスケジューラの実装
  - 統計処理パッケージRの並列化
  - 遺伝的アルゴリズムに特化したスケジューラ
- 大規模環境での実証実験