ソフトウェアアーキテクチャ

第8回　ネットワークシステム

環境情報学部

萩野　達也

lecture URL

https://vu5.sfc.keio.ac.jp/slide/

ソフトウェア

- 基本ソフトウェア
  - オペレーティングシステム
- 単体で動作するソフトウェア
  - シェル
  - コンパイラ
  - インタープリタ
- ネットワークを利用するソフトウェア
  - WEB
  - 電子メール
  - チャット
  - IP電話

分散システム

- 統合化コンピュータシステム用ソフトウェアを整備して，ネットワークによって結合された自律コンピュータの集合体
  - 「分散システム　ーコンセプトとデザイン」第2版 George Coulouris, Jean Dollimore, Tim Kindberg著, 水野忠則他訳，電気書院

分散システムの例

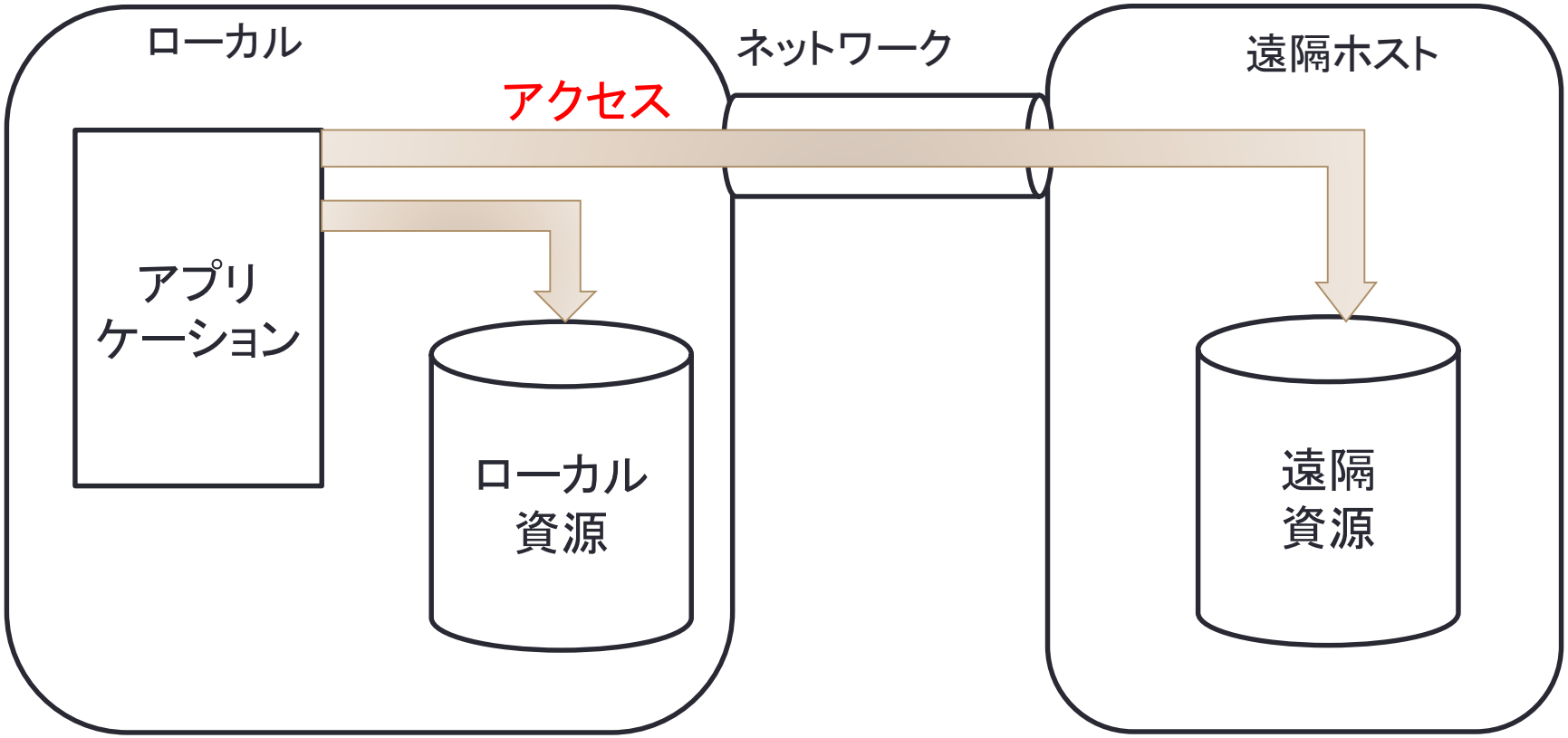
- ネットワークアプリケーション
  - 電子メール
  - 電子ニュース
  - World Wide Web
- 商用システム
  - 航空会社の座席予約および発券処理
  - 銀行の自動支払い機（ATM）
  - 在庫管理システム
- LANにつながれたPCなど
  - ファイル共有
  - プリンタ共有
  - 遠隔利用
- 遠隔会議アプリケーション
  - コンピュータ支援学習（e-learning）
  - 遠隔会議（H.323）
  - 共同設計作業（CSCW, Computer Supported Cooperative Work）

分散システムを考えるとき

- 資源の共有
  - どの資源を共有しているのか
  - だれが資源を持っているのか
- 開放性
  - だれでもが参加可能である
  - 閉鎖的なものは最初独占できるが，規模が大きくなれない
- 並列処理
  - 複数のことが同時に行われる
  - 同時に処理を要求されるかもしれない
- フォールトトレラント
  - 分散システムは複数のPCなどから構成される
  - すべてが故障なく動いている
- 透過性（Transparency）
  - 分散していることを感じさせない.
    - アクセス透過性
    - 位置透過性
    - 並行透過性
    - 複製透過性
    - 故障透過性
    - 移動透過性
    - 性能透過性
    - 規模透過性

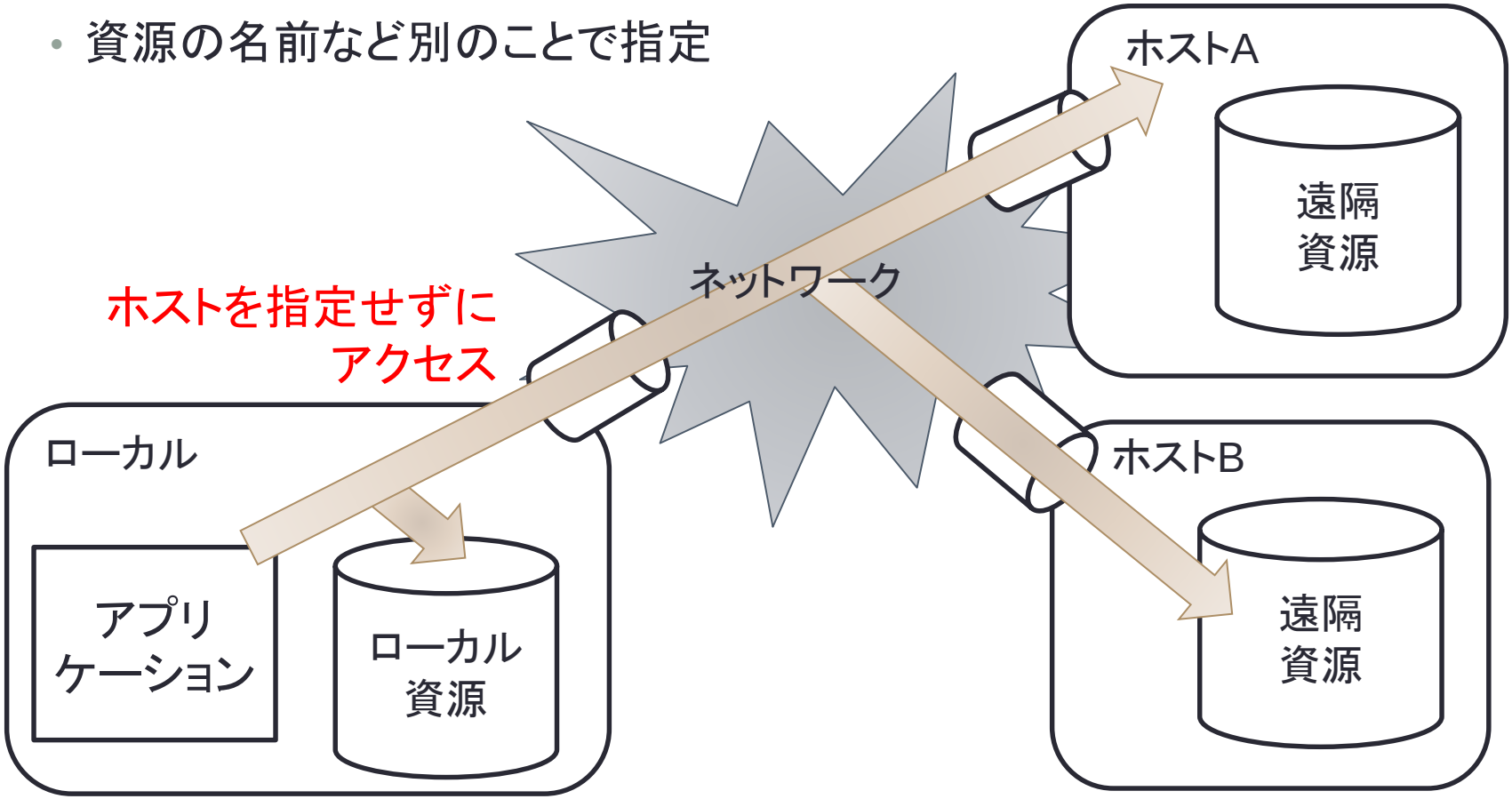
アクセス透過性

- ローカルと遠隔の資源を同じようにアクセス可能
  - 遠隔資源のアクセスに特別なことする必要はない



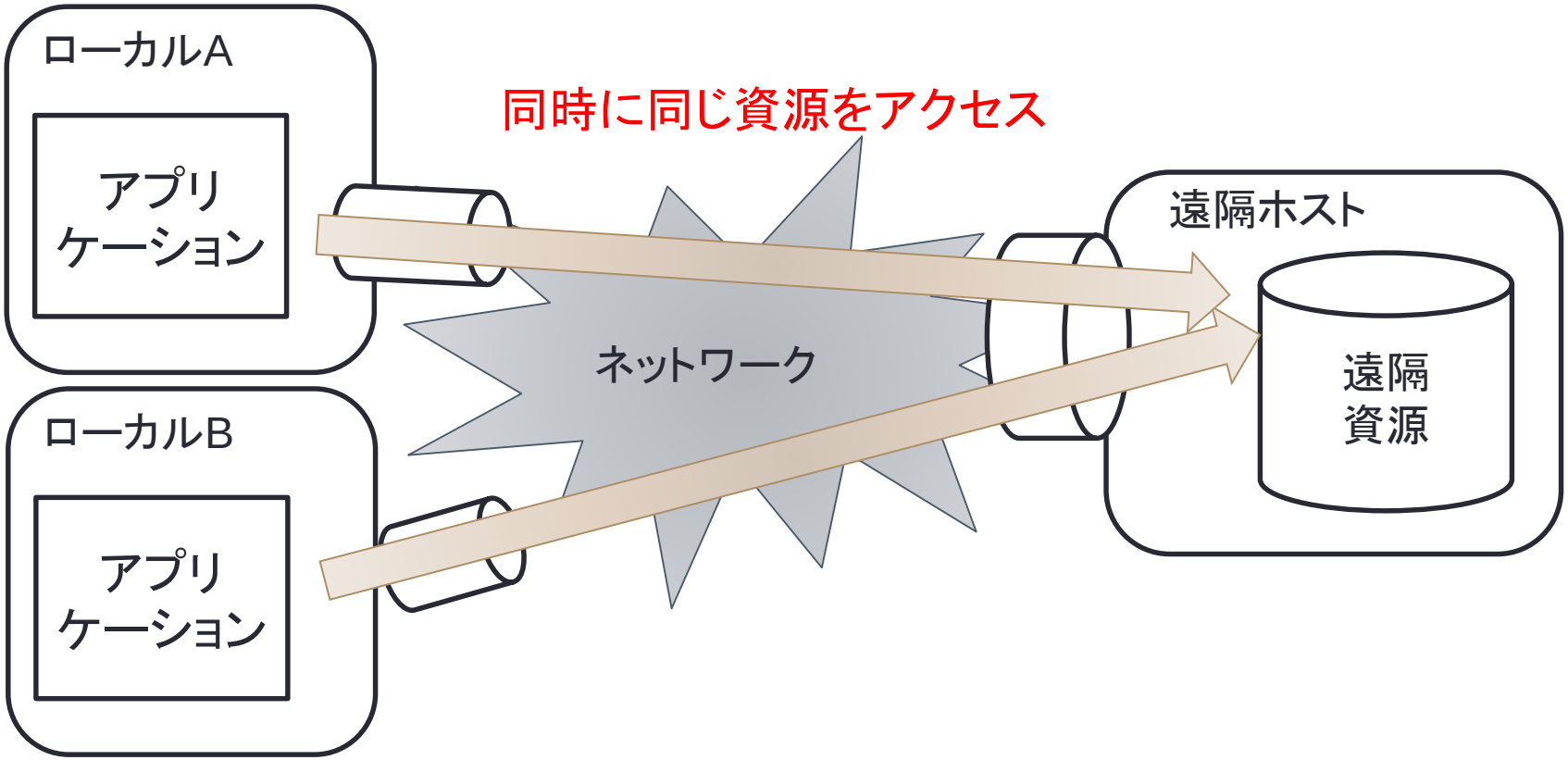
位置透過性

- 資源の位置に関する知識なしにアクセス可能
  - どこにあるかは知らなくても良い
  - 資源の名前など別のことで指定



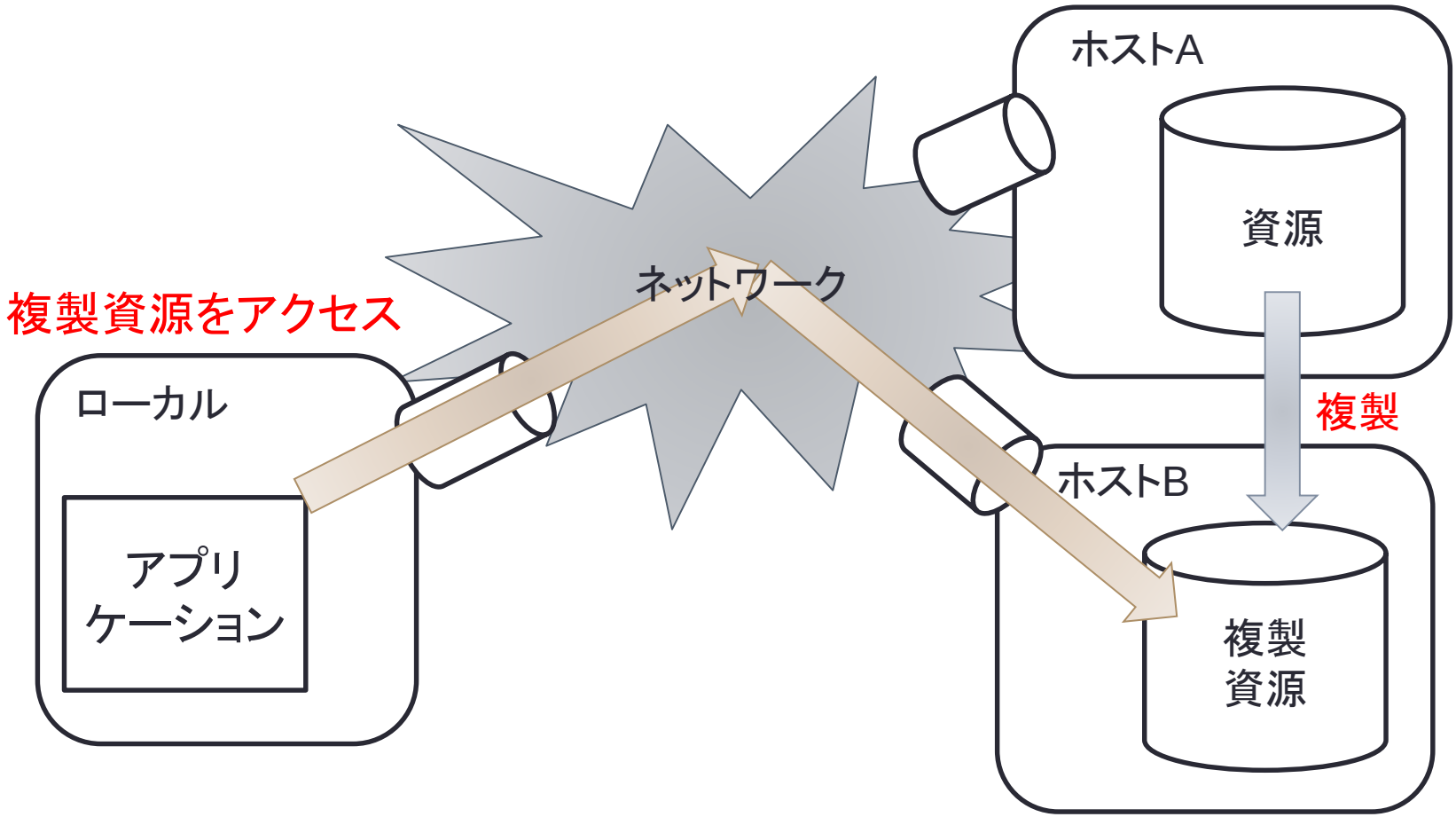
並行透過性

- 複数から同時に並行して操作が可能
  - だれかが利用中は使えないなどない



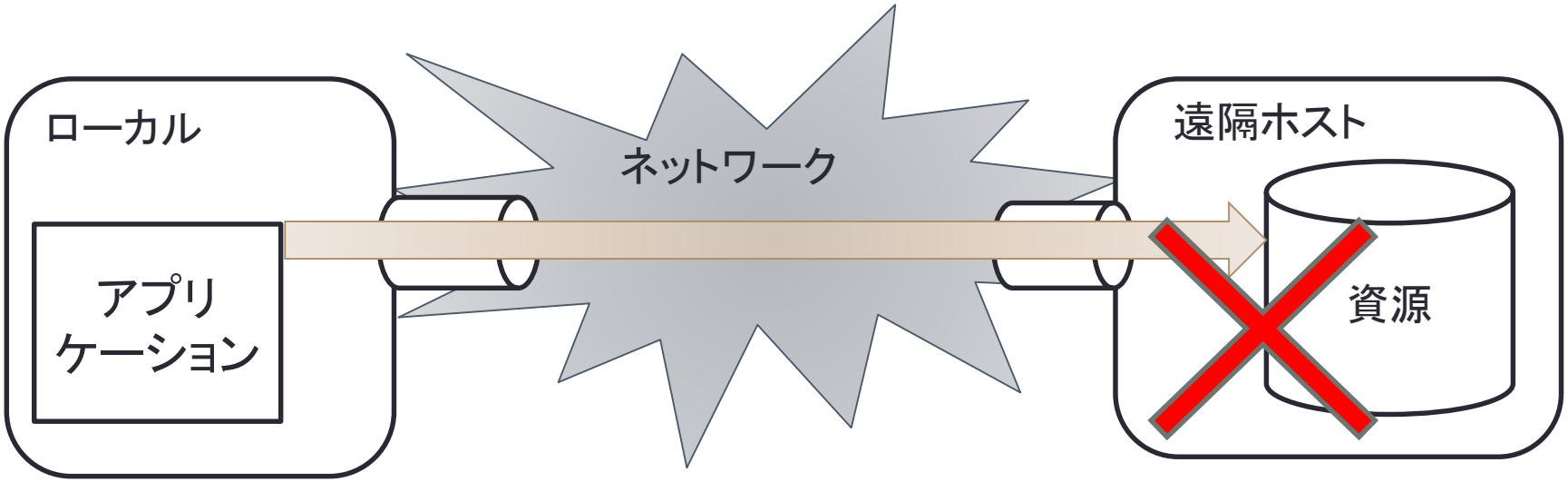
複製透過性

- 信頼性と性能を向上するために複製して処理可能



故障透過性

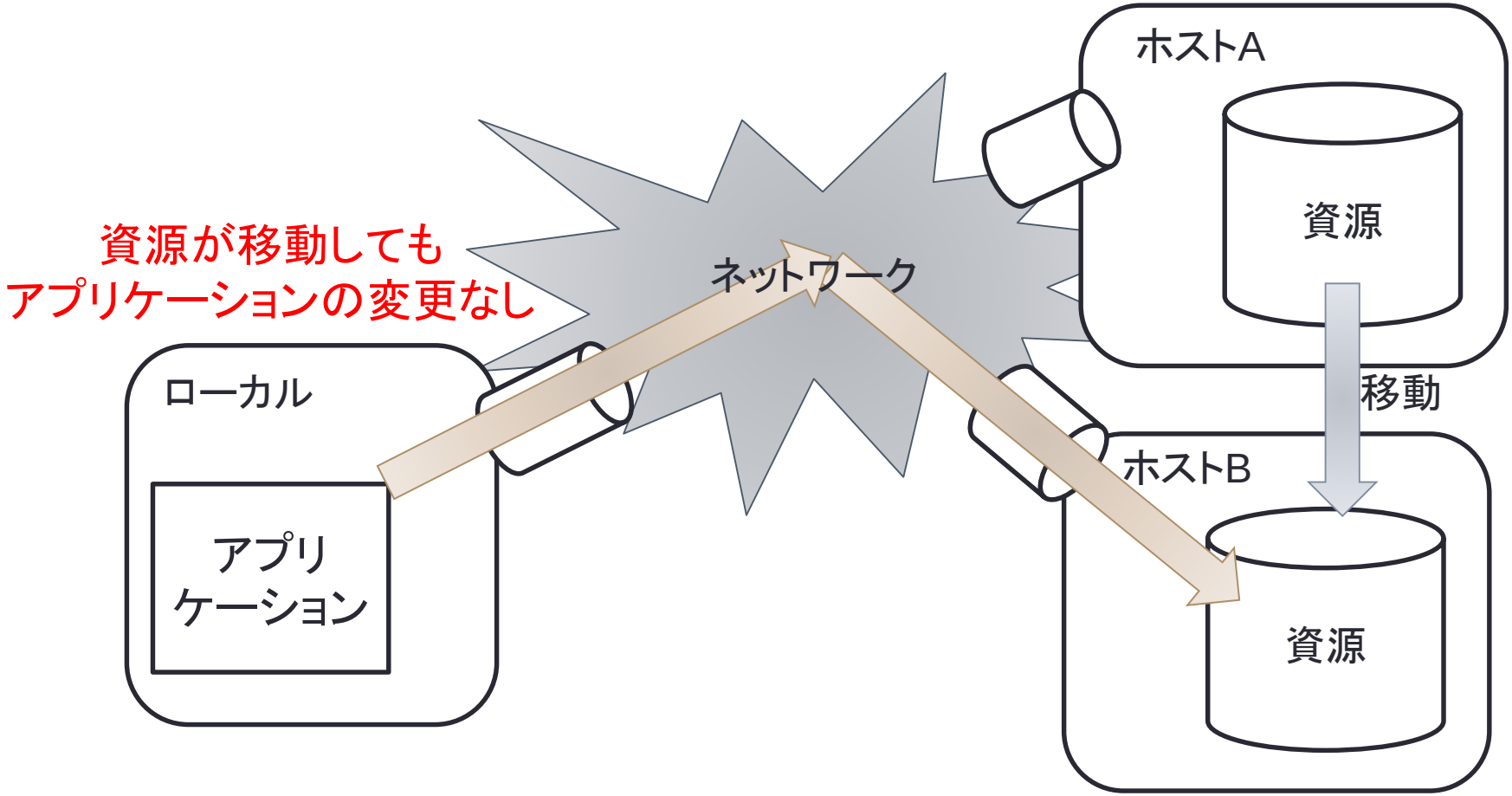
- ソフトウェアおよびハードウェアの故障にもかかわらず，故障を隠蔽して操作可能



ハード・ソフトの故障でも操作可能

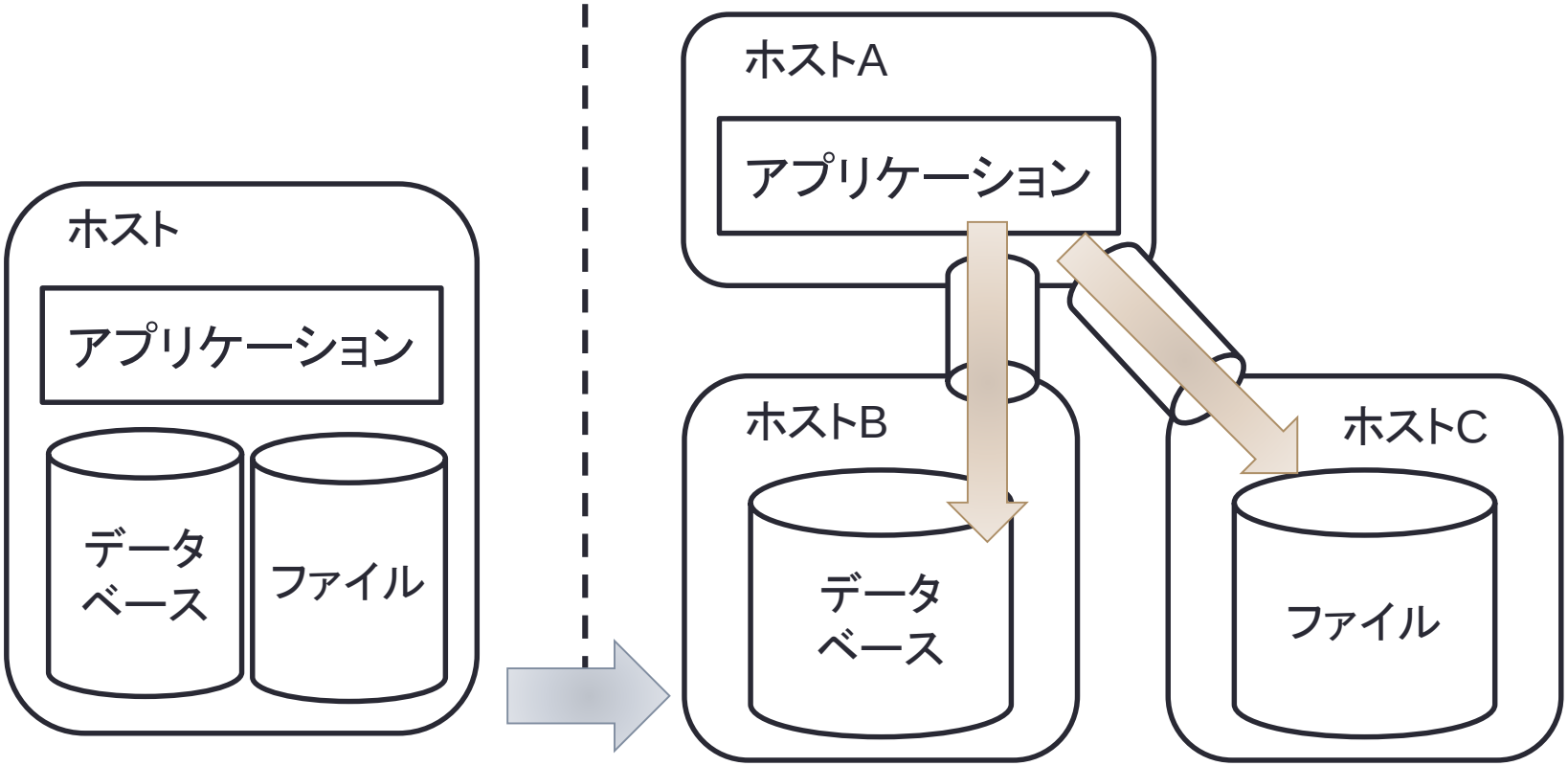
移動透過性

- 資源を移動させてもユーザおよびアプリケーションには影響を与えない



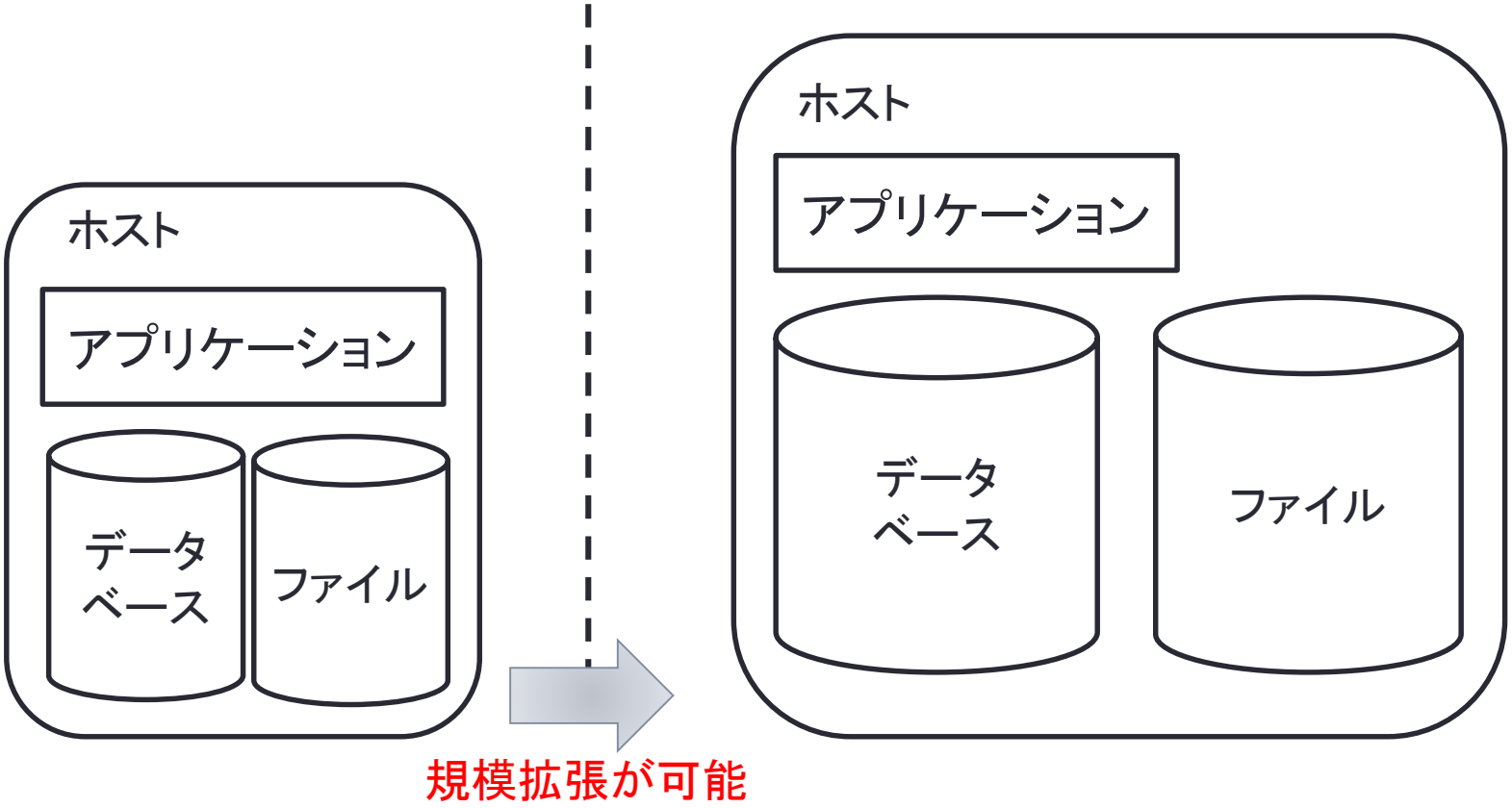
性能透過性

- システムの負荷が変化するに従い，性能を改善するための再構成が可能



規模透過性

- システム，アプリケーションの構造を変更することなく規模の拡張が可能



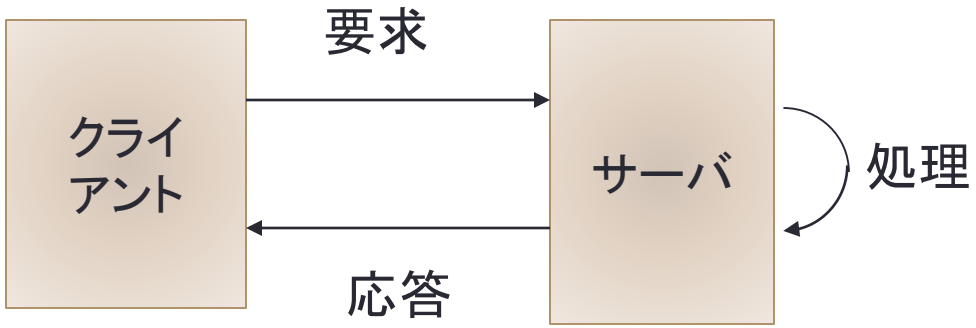
分散システムにおける通信

- 分散システムは分散しているため通信が必要

- 代表的な通信形態
  - クライアントサーバモデル
  - RPC
  - 関数移送（function shipping）
  - グループマルチキャスト
  - P2P

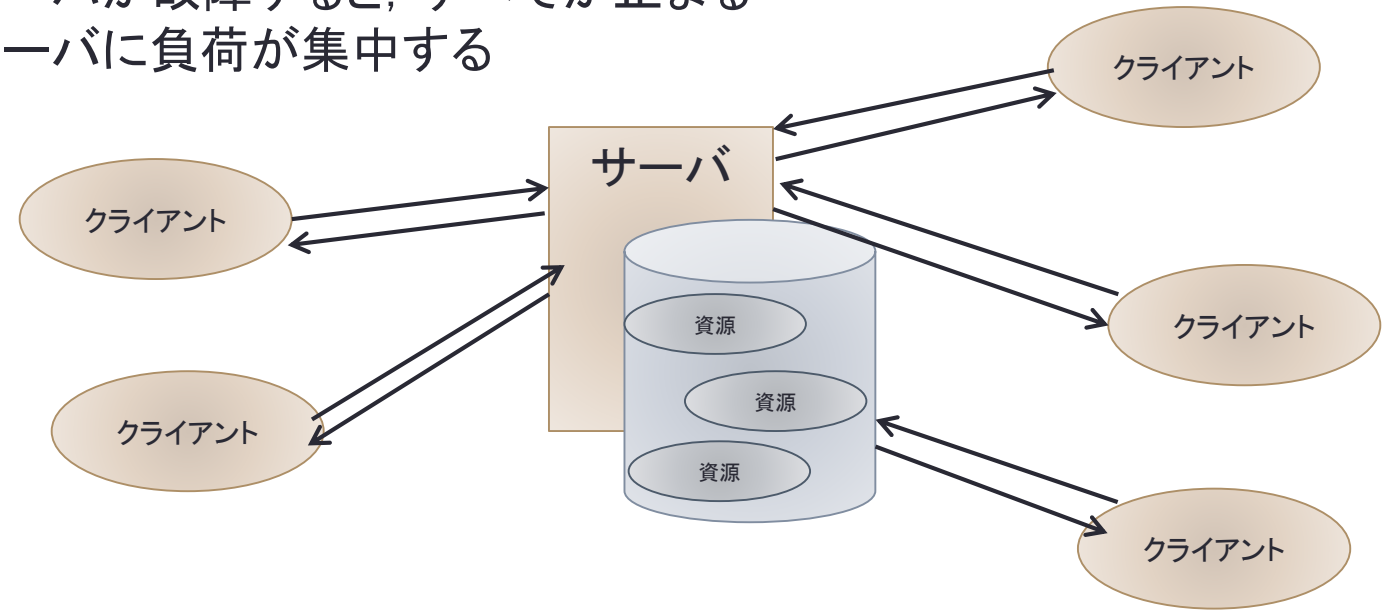
クライアントサーバモデル

- サーバ
  - サービスを提供する側
  - 資源を管理
- クライアント
  - サービスを要求する側
- 処理の流れ
  - クライアントからサーバに処理の要求を送信する
  - サーバはその要求を処理する
  - クライアントはサーバから処理の応答を受け取る



クライアントサーバモデルの問題点

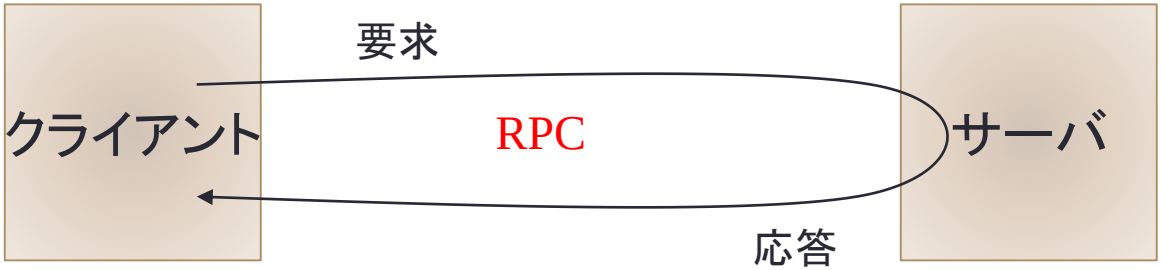
- 特徴
  - 資源の管理が簡単
  - 並列に同一資源の変更が要求された場合も，ロックなど行って処理を正しく行うことができる
- 問題点
  - サーバ中心
  - サーバが故障すると，すべてが止まる
  - サーバに負荷が集中する



RPC (Remote Procedure Call)

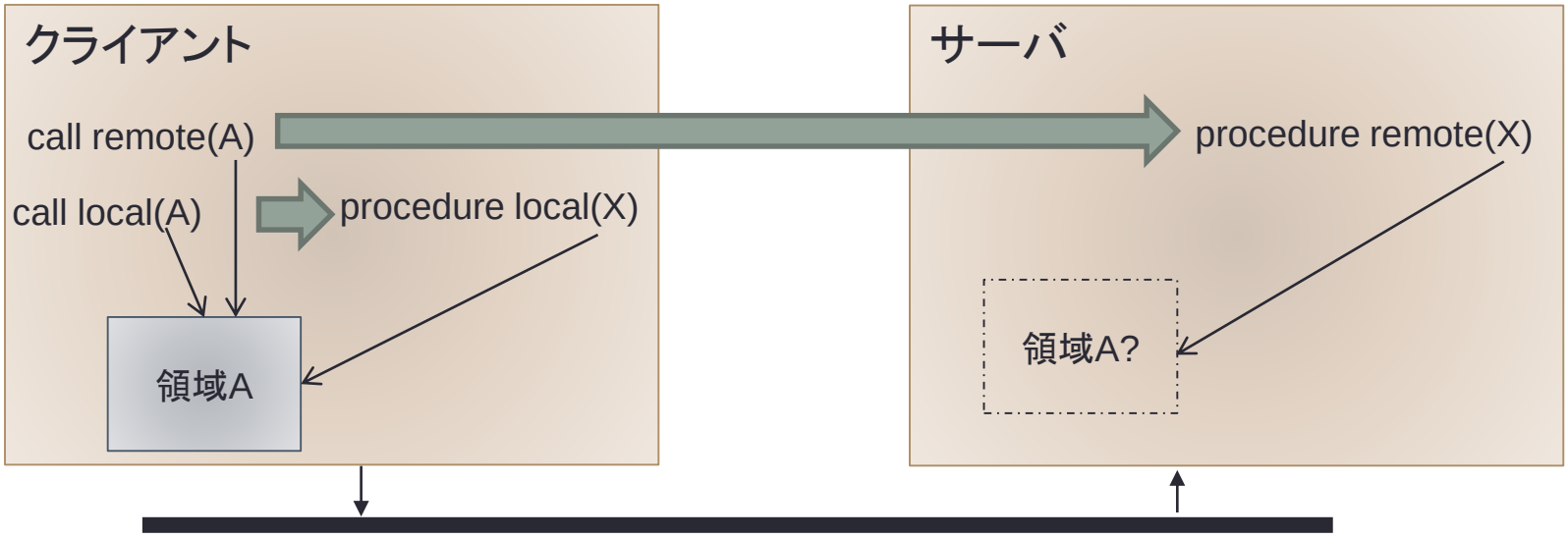
- クライアントサーバモデルの特殊なケース
  - 一般的には，サービスの要求と応答は必ずしも対応している必要はないが，
  - クライアントからの一つのサービスの要求に対して，その処理が終了した後に応答として結果を返すことが多い．
  - クライアントはサーバからの応答が返ってくるまで待つ．
  - サーバにある手続きを呼び出しているようにみなすことができる．

- RPC（Remote Procedure Call）
  - 遠隔手続き呼び出し
  - クライアントプログラム的には通常の手続き呼び出しと同じ
  - 違いは，手続きがローカルにあるか遠隔のサーバにあるか．
  - 下位プロトコルを最適化することも可能（応答により要求の到達を確認）



RPCとローカル手続きの違い(1)

- ポインタを渡すことはできない
  - ローカル手続きではポインタで大きな領域などの参照を渡すことができる(call-by-reference)
  - RPCではクライアントとサーバのメモリ空間が異なるために，クライアントのポインタはサーバでは意味がない
  - call-by-valueでなくてはならない.



RPCとローカル手続きの違い(2)

- 処理途中での障害を考慮しなくてはならない
  - 要求メッセージを再送するのか？
    - 要求がサーバに届いていないかもしれない.
  - 要求メッセージの重複を削除するのか？
    - 要求が再送される場合，重複を削除しないと，二重に実行されるかもしれない.
  - 応答を再送するのか？
    - 応答がクライアントに帰っていないかもしれない.
    - 再送するためには応答を残しておく必要がある

RPCのセマンティックス

- 不確実呼び出し (maybe call)
  - 要求の再送は行わない
    - 要求の再送はない → 重複の削除は必要ない
  - 応答の再送は行わない

- 最低一度呼び出し(at-least-once call)
  - 要求メッセージの再送を行う
  - サーバ側で重複はチェックしない
  - クライアントは応答が返るまで要求の再送を行う
  - サーバは処理を少なくとも一度は行う
  - idempotentな処理に向く

- 最大一度呼び出し(at-most-once call)
  - 要求メッセージの再送を行う
  - サーバ側で重複のチェックを行う
  - サーバは処理を多くとも一度しか行わない
  - トランザクション処理などはこのタイプ

RPCの実装

- RPCの実装の下の部分は共通化できる
  - クライアントにおける手続き呼び出しの情報をパケットにする
  - サーバにおいてパケットを解析して手続きを呼び出す
  - 結果をクライアントに返す
  - RPCの内容ではなく，インターフェイス（引数と戻り値の型）にのみ依存
- インターフェイス定義言語
  - RPCの入出力パラメータを指定
  - stubの自動生成
    - stub=切り株
    - 手続き呼び出しをパケットにする
    - パケットを解析して対応する手続きを呼び出す

RPCのインターフェイスの記述

- SunRPCによるインターフェイスの記述

add.x

```
struct intpair { int a; int b; };
program ADD {
  version ADDVARS {
    int PRINT(string) = 1; /* procedure number = 1 */
    int ADD(intpair) = 2;  /* procedure number = 2 */
  } = 5;                   /* version number = 5 */
} = 0x20000099;            /* program number = 0x20000099 */
```

rpcgen -C add.x

add_clnt.c

```
...
int *print_5(char **argp, CLIENT *clnt)
{
  static int clnt_res;

  memset((char *)&clnt_res, 0, sizeof (clnt_res));
  if (clnt_call(clnt, PRINT,
     (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
     (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
     TIMEOUT) != RPC_SUCCESS) {
       return (NULL);
  }
  return (&clnt_res);
}
```

add_svc.c

```
...
add_5(struct svc_req *rqstp, SVCXPRT *transp)
{
  union {
    char *print_5_arg;
    intpair add_5_arg;
  } argument;
  char *result;
  ...
  switch (rqstp->rq_proc) {
  case PRINT:
      xdr_argument = (xdrproc_t) xdr_wrapstring;
      xdr_result = (xdrproc_t) xdr_int;
      local = (char *(*)(char *, struct svc_req
*)) print_5_svc;
       break;
```

RPCの特徴と問題点

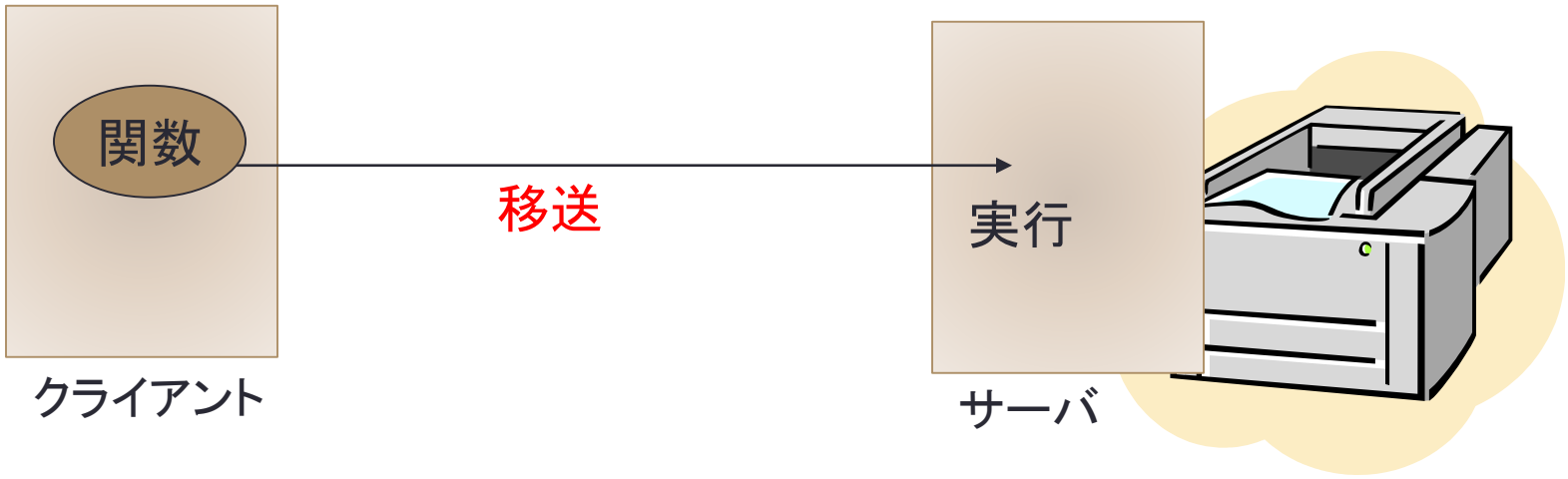
- 特徴
  - プログラムからはローカルな手続きと同じ感覚で利用できる
  - インターフェイスからstubが自動生成される

- 問題点
  - 処理途中での故障について考慮する必要がある
  - 前もってインターフェイスを決めておく必要がある
    - 存在しない手続きは呼び出すことができない
  - 一つ一つの手続きを呼び出す必要がある
    - サーバ側で組み合わせることはできない

関数移送 (Function Shipping)

- クライアントからサーバに処理の命令群を送り処理する.
  - 特定の手続きに限定されない
  - 複数の処理をまとめて命令群にして送ることができる
  - サーバは特別な命令群を処理できるインタプリタ

- 例
  - ポストスクリプトプリンタ
  - NeWSウィンドウシステム（ディスプレイポストスクリプト）



ポストスクリプト（PostScript）

- ページ記述言語
  - 描画命令
  - スタック指向プログラミング言語
  - 逆ポーランド記法を用いる

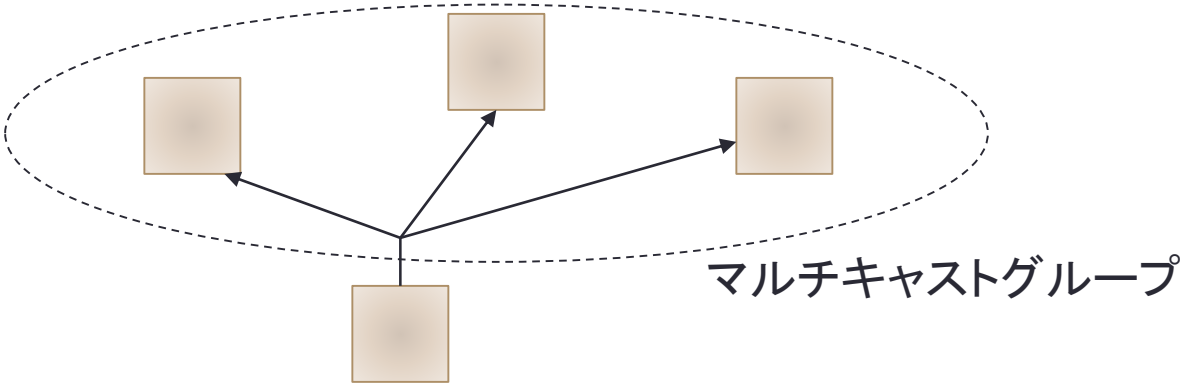
```
%!
% macro (draw rectangle) ; usage: left top width height RRECT
/RRECT { newpath 4 copy pop pop moveto dup 0 exch rlineto exch 0 rlineto
neg 0 exch rlineto closepath pop pop } def

100 100 100 150 RRECT
.5 setgray
fill

100 300 moveto
/Helvetica findfont
12 scalefont
setfont
.5 0 .5 0 setcmykcolor
(test string) show

showpage
```

クライアントサーバの非対称性

- サーバが資源管理を一手に引き受ける
  - 実装が容易
  - サーバがボトルネックになる
  - サーバのセキュリティホールが命取り
- サーバの数　＜＜＜　クライアントの数
- サーバが巨大になる
  - 持ち運ぶのはクライアント
  - サーバと通信できなければ，仕事はできない

グループマルチキャスト

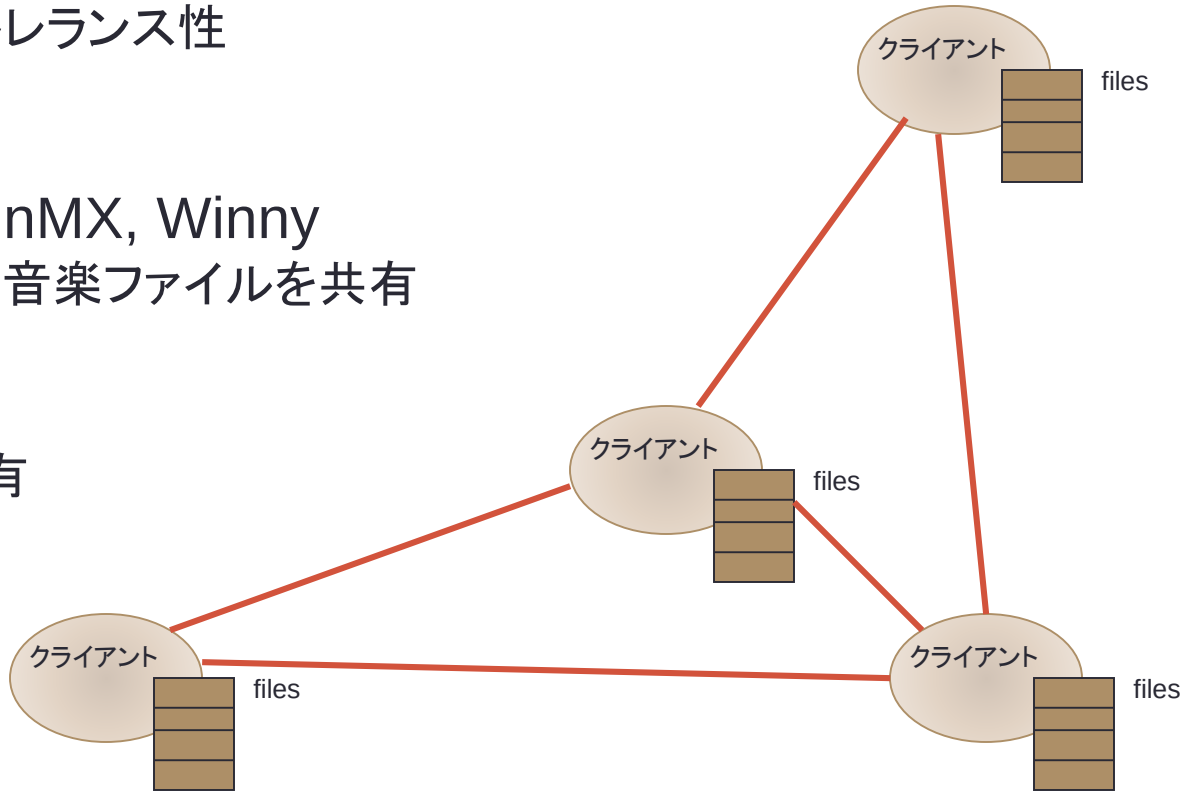
- 単一の相手に通信するのではなく，グループにマルチキャストする



- オブジェクトの位置の特定
  - マルチキャストし対応するサーバのみが答える
- 複数同時更新
  - 同時に更新可能
- フォールトトレランス性
  - グループで応答する場合，1台が故障しても大丈夫

P2P (Peer to Peer)

- クライアントサーバのように非対称ではない
- クライアント同士が直接通信する
  - クライアントがサーバを兼ねる
  - フォールトトレランス性
  - 匿名性
- Napster, WinMX, Winny
  - MP3などの音楽ファイルを共有
- Gnutella
  - ファイル共有



まとめ

- 分散システム
- 透過性
  - アクセス透過性
  - 位置透過性
  - 並行透過性
  - 複製透過性
  - 故障透過性
  - 移動透過性
  - 性能透過性
  - 規模透過性
- 通信モデル
  - クライアントサーバモデル
  - RPC
  - 関数移送
  - グループマルチキャスト
  - P2P