

メッセージ交換



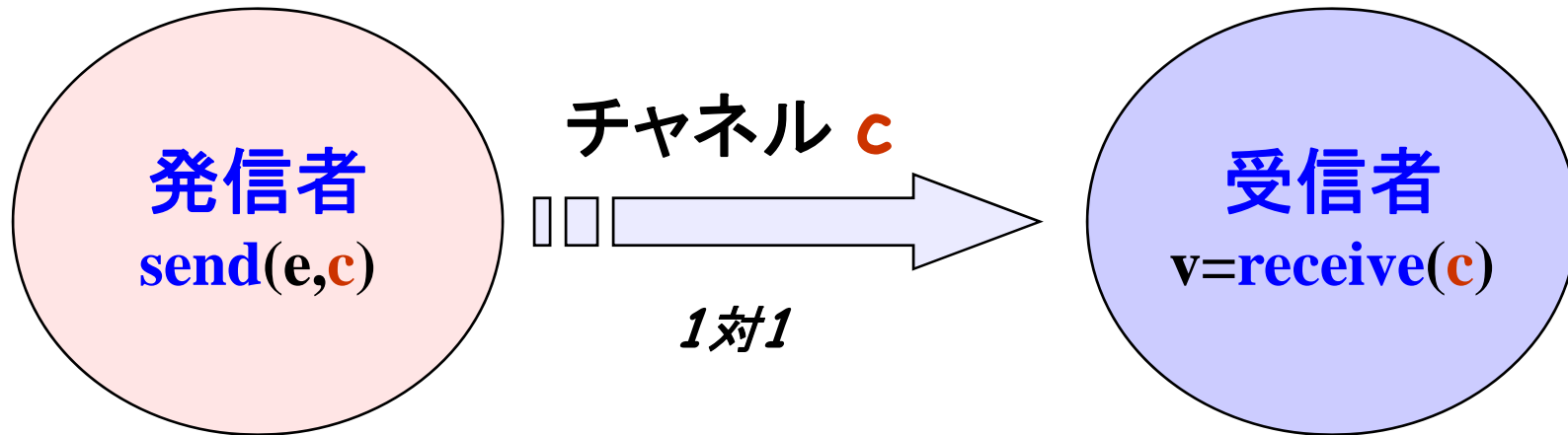
メッセージ交換

概念: 同期的 メッセージ交換 - チャンネル
非同期的 メッセージ交換 - ポート
- send と receive / selective receive
待合わせ 双方向通信 - エントリ
- call と accept ... reply

モデル: チャンネル : ラベルの付け替え, 選択 と ガード
ポート : メッセージ待ち行列, 選択 と ガード
エントリ : ポート と チャンネル

実践: 分散計算 (個別記憶)
スレッド と モニタ (共有記憶)

10.1 同期的メッセージ交換 – チャンネル



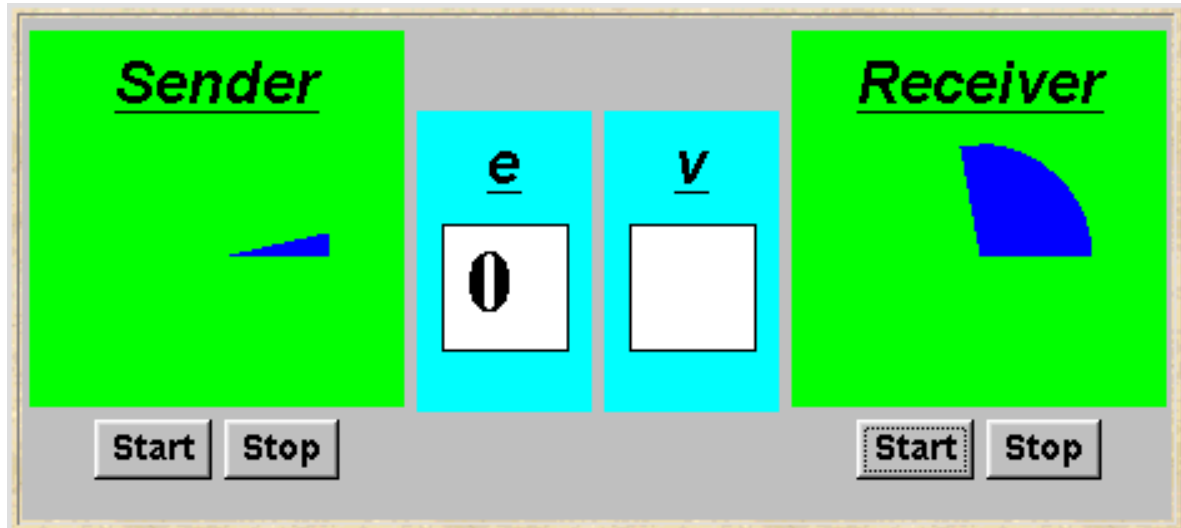
◆ `send(e, c)` - 式 e の値を、チャンネル c に送る。送信操作を呼んだプロセスは、メッセージがチャンネルから受け取られるまで **ブロック** される。

◆ `v = receive(c)` - チャンネル c から受け取った値を局所変数 v にしまう。受信操作を呼んだプロセスは、メッセージがチャンネルに送られるまで **ブロック** 待ちになる。

同期的メッセージ交換 - アプレット

送信者は受信者と
単一の **チャンネル**を
通して通信する。

送信者は整数を0
から9の順に送り、
その後ふたたび0
から送信していく。



```
Channel chan = new Channel();  
tx.start(new Sender(chan, senddisp));  
rx.start(new Receiver(chan, recvdisp));
```

ThreadPoolのインスタンス

SlotCanvasのインスタンス

Java による実装 – channel

```
class Channel extends Selectable {
    Object chann = null;

    public synchronized void send(Object v)
        throws InterruptedException {
        chann = v;
        signal();
        while (chann != null) wait();
    }

    public synchronized Object receive()
        throws InterruptedException {
        block(); clearReady(); //part of Selectable
        Object tmp = chann; chann = null;
        notifyAll(); //could be notify()
        return(tmp);
    }
}
```

Channel の実装は `send` と `receive` という同期的アクセスメソッドをもつモニタである。

Selectable については後述。

Java による実装 - sender

```
class Sender implements Runnable {
    private Channel chan;
    private SlotCanvas display;
    Sender(Channel c, SlotCanvas d)
        {chan=c; display=d;}

    public void run() {
        try { int ei = 0;
            while(true) {
                display.enter(String.valueOf(ei));
                ThreadPanel.rotate(12);
                chan.send(new Integer(ei));
                display.leave(String.valueOf(ei));
                ei=(ei+1)%10; ThreadPanel.rotate(348);
            }
        } catch (InterruptedException e){}
    }
}
```

Java による実装 - receiver

```
class Receiver implements Runnable {
    private Channel chan;
    private SlotCanvas display;
    Receiver(Channel c, SlotCanvas d)
        {chan=c; display=d;}

    public void run() {
        try { Integer v=null;
            while(true) {
                ThreadPanel.rotate(180);
                if (v!=null) display.leave(v.toString());
                v = (Integer)chan.receive();
                display.enter(v.toString());
                ThreadPanel.rotate(180);
            }
        } catch (InterruptedException e){}
    }
}
```

モデル

```
range M = 0..9 // messages with values up to 9

SENDER = SENDER[0], // shared channel chan
SENDER[e:M] = (chan.send[e]-> SENDER[(e+1)%10]).

RECEIVER = (chan.receive[v:M]-> RECEIVER).

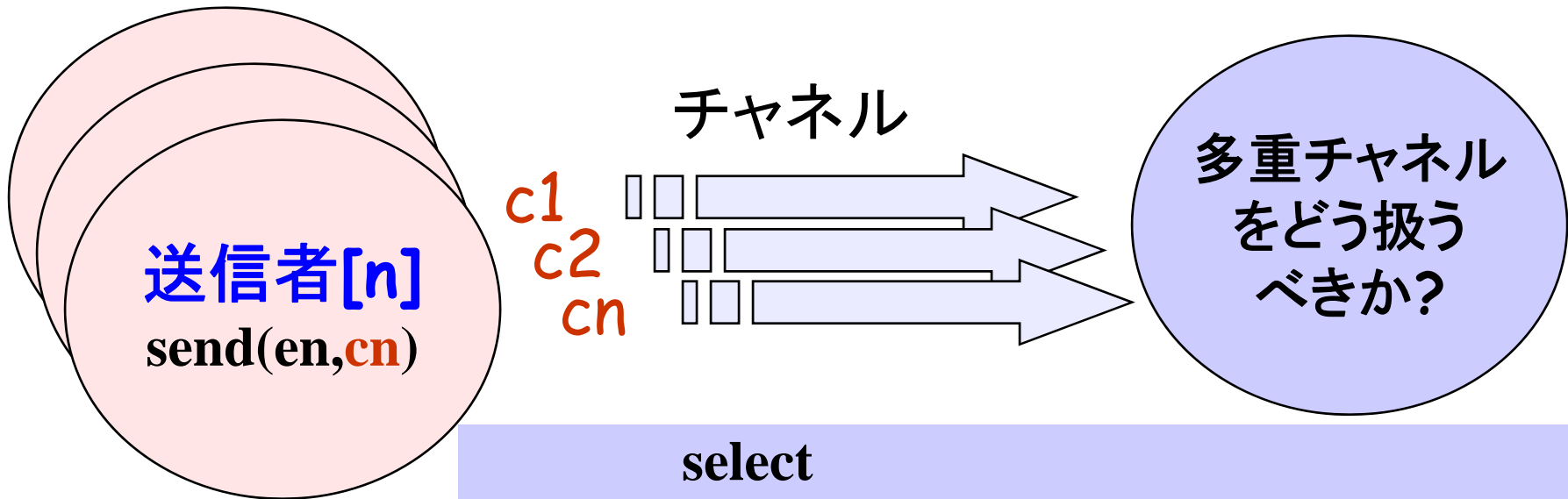
// relabeling to model synchronization
|| SyncMsg = (SENDER || RECEIVER)
/ {chan/chan.{send, receive}}.
```

LTS?

これをラベル付け替え
を必要としないように直
接モデル化するには、
どうすればよいか?

メッセージ操作	FSP モデル
<code>send(e,chan)</code>	<code>chan.[e]</code>
<code>v = receive(chan)</code>	<code>chan.[v:M]</code>

選択的受信



Select 文...

これをFSPでいかにモデル化するか?

```
select
```

```
  when  $G_1$  and  $v_1 = \text{receive}(\text{chan}_1) \Rightarrow S_1;$ 
```

```
or
```

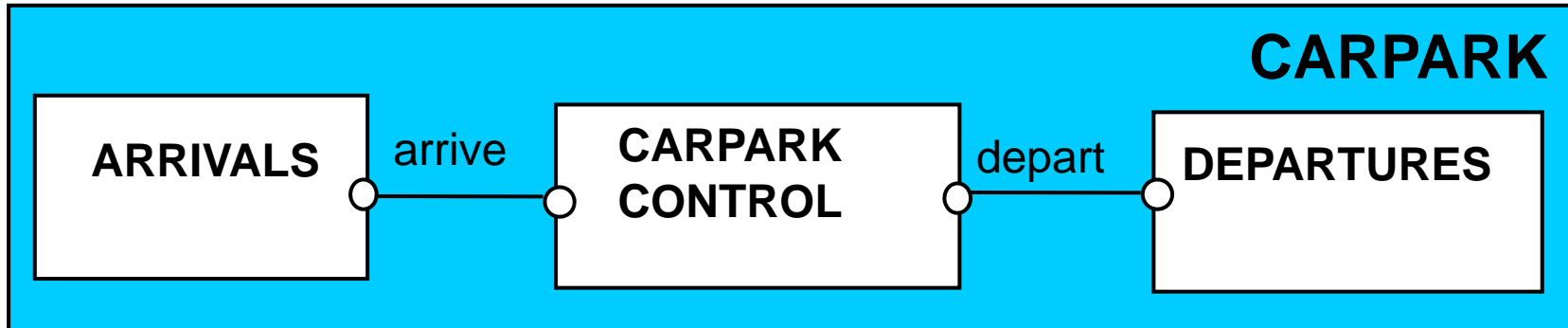
```
  when  $G_2$  and  $v_2 = \text{receive}(\text{chan}_2) \Rightarrow S_2;$ 
```

```
or
```

```
  when  $G_n$  and  $v_n = \text{receive}(\text{chan}_n) \Rightarrow S_n;$ 
```

```
end
```

選択的受信



```
CARPARKCONTROL(N=4) = SPACES[N],  
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]  
                  |when(i<N) depart->SPACES[i+1]  
                  ).
```

```
ARRIVALS = (arrive->ARRIVALS).  
DEPARTURES = (depart->DEPARTURES).
```

```
|| CARPARK = (ARRIVALS || CARPARKCONTROL(4)  
             || DEPARTURES).
```

チャンネルとして解釈

メッセージ交換による実装は？

Java による実装 – 選択的受信

```
class MsgCarPark implements Runnable {
    private Channel arrive,depart;
    private int spaces,N;
    private StringCanvas disp;

    public MsgCarPark(Channel a, Channel l,
                      StringCanvas d, int capacity) {
        depart=l; arrive=a; N=spaces=capacity; disp=d;
    }
    ...
    public void run() {...}
}
```

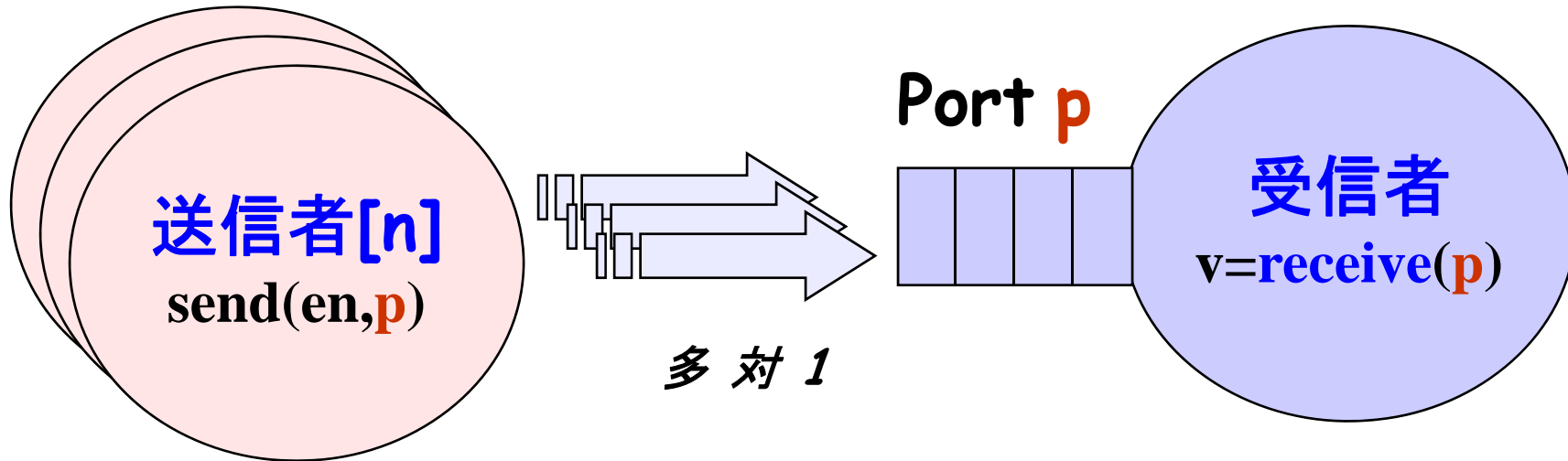
CARPARKCONTROLをチャネル arriveと depart から信号を受け取るスレッド MsgCarParkとして実装.

Java による実装 – 選択的受信

```
public void run() {
    try {
        Select sel = new Select();
        sel.add(depart);
        sel.add(arrive);
        while(true) {
            ThreadPanel.rotate(12);
            arrive.guard(spaces>0);
            depart.guard(spaces<N);
            switch (sel.choose()) {
                case 1:depart.receive();display(++spaces);
                    break;
                case 2:arrive.receive();display(--spaces);
                    break;
            }
        }
    } catch InterruptedException{}
}
```

アプレットを
見よ

10.2 非同期的メッセージ交換 – ポート



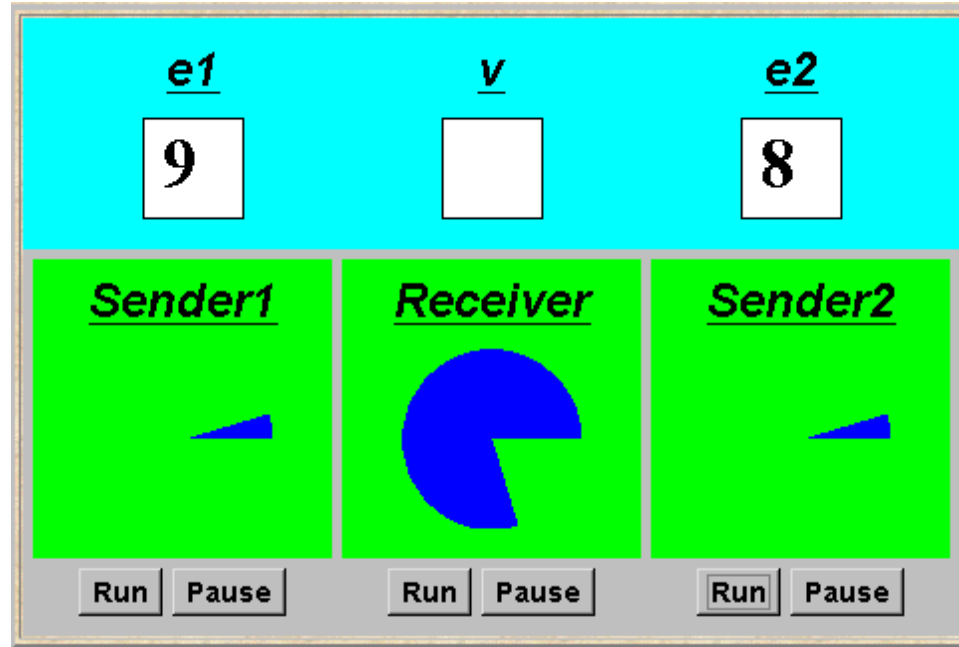
◆ $\text{send}(e, p)$ - 式 e の値をポート p に送る. 送信操作を呼ぶプロセスはブロックされない. 受信者が待ちでないときは, メッセージはポートの待ち行列に入れられる.

◆ $v = \text{receive}(p)$ - 値をポート p から受け取り局所変数 v にしまう. 受信操作を呼ぶプロセスは, ポートの待ち行列にメッセージがないときはブロックされる.

非同期的メッセージ交換 - アプレット

2人の送信者が1人の受信者と“無限の” **ポート**を介して通信している。

各送信者は整数を0から9の順に送り、その後ふたたび0から送信していく。



```
Port port = new Port();  
tx1.start(new Asender(port, send1disp));  
tx2.start(new Asender(port, send2disp));  
rx.start(new Areceiver(port, recvdisp));
```

ThreadPanel1のインスタンス

SlotCanvas のインスタンス

Java による実装 - ポート

```
class Port extends Selectable {
    Vector queue = new Vector();

    public synchronized void send(Object v){
        queue.addElement(v);
        signal();
    }

    public synchronized Object receive()
        throws InterruptedException {
        block(); clearReady();
        Object tmp = queue.elementAt(0);
        queue.removeElementAt(0);
        return(tmp);
    }
}
```

Port の実装は `send` と `receive` 用の同期的アクセスメソッドを持つモニタである。

ポートモデル

```
range M = 0..9           // messages with values up to 9
set S = {[M], [M][M]}   // queue of up to three messages

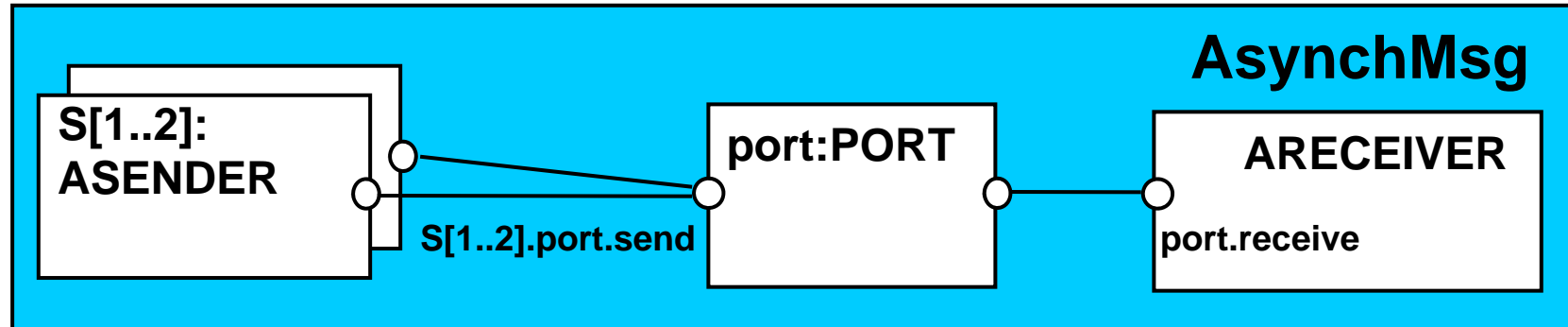
PORT                       //empty state, only send permitted
  = (send[x:M]->PORT[x]),
PORT[h:M]                  //one message queued to port
  = (send[x:M]->PORT[x][h]
    | receive[h]->PORT
    ),
PORT[t:S][h:M]            //two or more messages queued to port
  = (send[x:M]->PORT[x][t][h]
    | receive[h]->PORT[t]
    ).

// minimise to see result of abstracting from data values
| | APORT = PORT / {send/send[M], receive/receive[M]}.
```

LTS?

4つの値を送ったらどうなるか?

アプレットのモデル



```
ASENDER = ASENDER[0],
```

```
ASENDER[e:M] = (port.send[e]->ASENDER[(e+1)%10]).
```

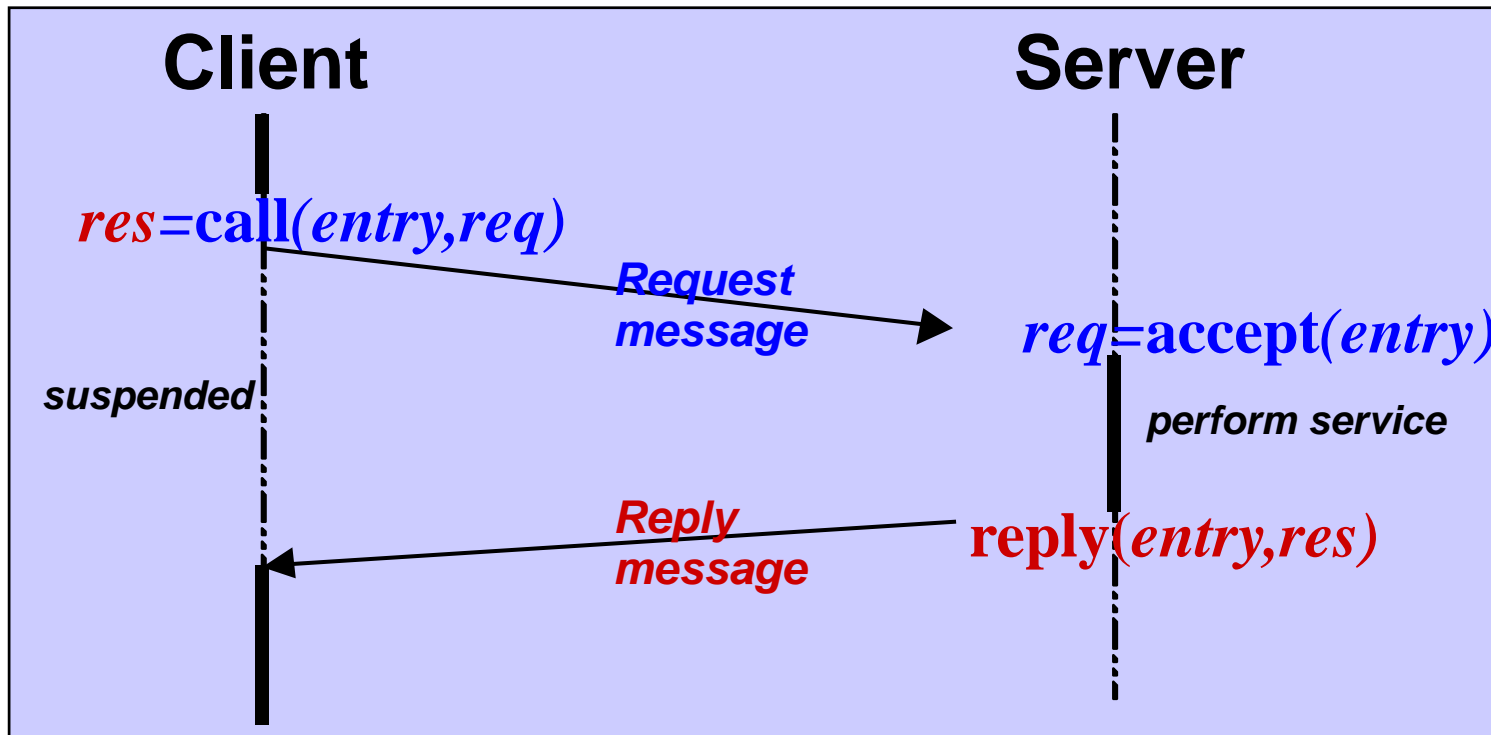
```
ARECEIVER = (port.receive[v:M]->ARECEIVER).
```

```
|| AsyncMsg = (s[1..2]:ASENDER || ARECEIVER | port:PORT)  
/ {s[1..2].port.send/port.send}.
```

安全性?

10.3 待ち合せ - エントリ

待ち合せは クライアント サーバ 通信を支援する要求-応答形式の一つである. 多くのクライアントがサービスを要求しうるが, 一時には1人だけがサービスを受けられる.



待ち合せ

◆ $res = call(e, req)$ - 値 req を要求メッセージとして送信し, それがエントリ e の待ち行列に入れられる.

◆ 呼び出し側のプロセスは, 応答メッセージが受け取られ局所変数 res に入れられるまでブロックされる.

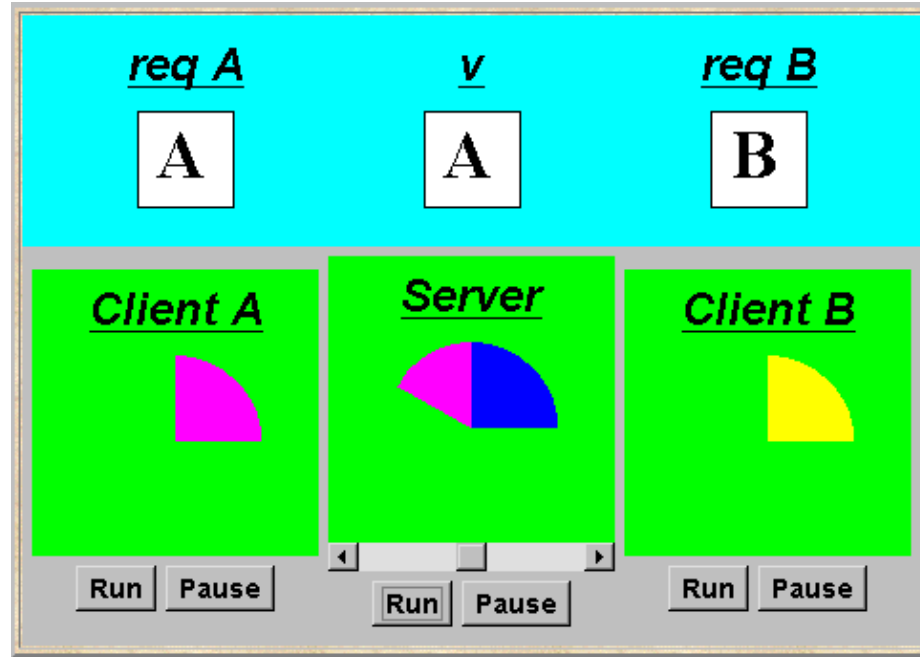
◆ $req = accept(e)$ - エントリ e から要求メッセージの値を受け取り局所変数 req にしまう. 呼び出し側のプロセスはエントリにメッセージがなければブロックされる.

◆ $reply(e, res)$ - 値 res を応答メッセージとしてエントリ e に送る.

このモデルと実装は, 一方向にポートを, 他方向にチャネルを使っている. **どっちがどっちか?**

非同期的メッセージ交換 - アプレット

2人のクライアントが一時には1つの要求にのみサービスするサーバを呼び出す。



```
Entry entry = new Entry();  
clA.start(new Client(entry, clientAdisp, "A"));  
clB.start(new Client(entry, clientBdisp, "B"));  
sv.start(new Server(entry, serverdisp));
```

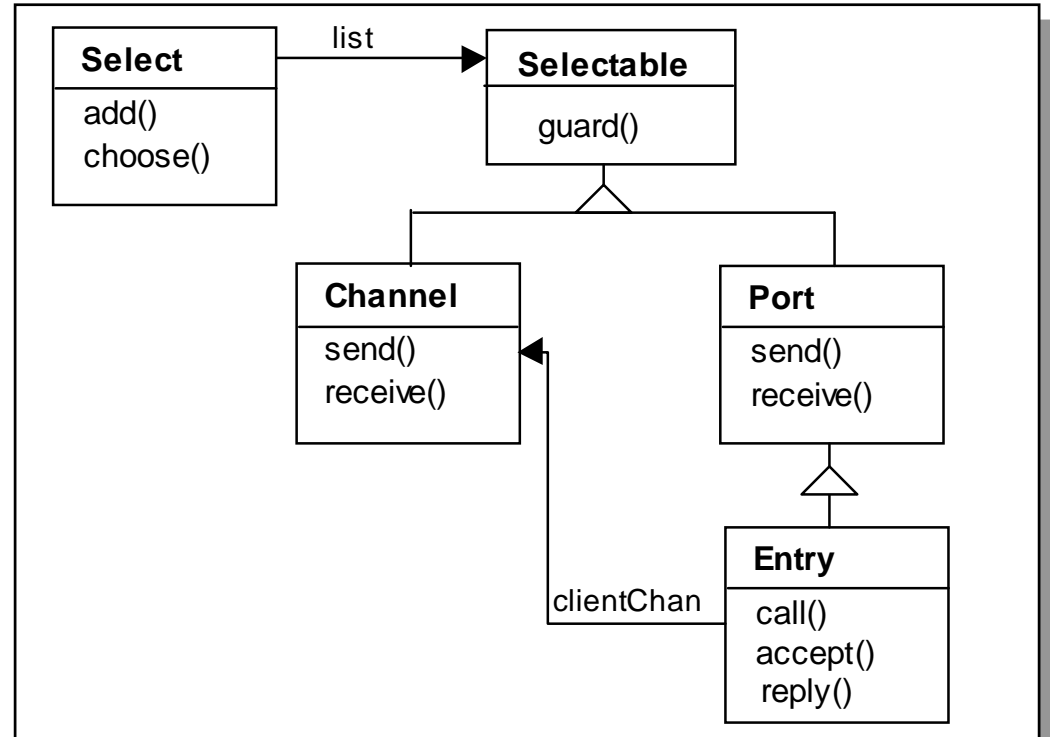
ThreadPanel1のインスタンス

SlotCanvasのインスタンス

Java による実装 - エントリ

エントリはポートを継承して実装され、したがって待ち行列と選択的受信をサポートする。

call メソッドは応答メッセージを受け取るチャネル・オブジェクトを創りだす。チャネルは自分への参照とreqオブジェクトへの参照からなるメッセージを構成してエントリに送る。そしてチャネルに出される応答を待つ。



acceptメソッドはチャネルの参照の複写を保持する。**reply**メソッドはこのチャネルに応答を送る。

Java による実装 - エントリ

```
class Entry<R,P> extends Port<R> {
    private CallMsg<R,P> cm;
    private Port<CallMsg<R,P>> cp = new Port<CallMsg<R,P>>();

    public P call(R req) throws InterruptedException {
        Channel<P> clientChan = new Channel<P>();
        cp.send(new CallMsg<R,P>(req,clientChan));
        return clientChan.receive();
    }

    public R accept() throws InterruptedException {
        cm = cp.receive();
        return cm.request;
    }

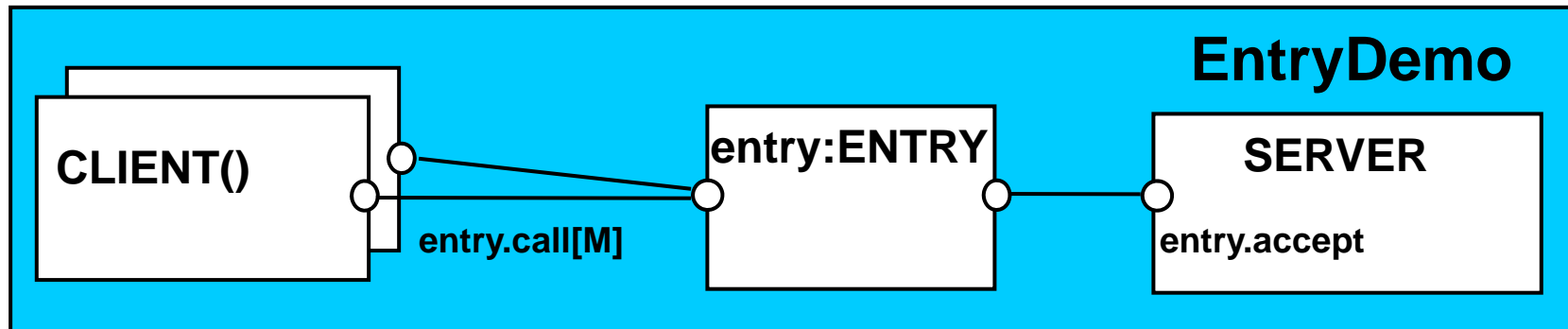
    public void reply(P res) throws InterruptedException {
        cm.replychan.send(res);
    }

    private class CallMsg<R,P> {
        R request;
        Channel<P> replychan;
        CallMsg(R m, Channel<P> c)
            {request=m; replychan=c;}
    }
}
```

call, accept, replyは同期メソッドでなければならぬか?

エントリとアプレットのモデル

ポートとチャネルのモデルを再利用する ...



```
set M = {replyA,replyB}           // reply channels
|| ENTRY = PORT/{call/send, accept/receive}.
CLIENT(CH='reply) = (entry.call[CH]->[CH]->CLIENT).
SERVER = (entry.accept[ch:M]->[ch]->SERVER).
|| EntryDemo = (CLIENT('replyA) || CLIENT('replyB)
                || entry:ENTRY || SERVER ).
```

式やパラメータに
使われる動作ラ
ベルは、引用符
を接頭辞としてつ
けなければならない。

待ち合せ 対 モニタ・メソッド起動

違いは何か?

- ... クライアントの視点から?
- ... サーバの視点から?
- ... 相互排除?

どちらの実装が効率的か?

- ... 局所的文脈では (クライアントとサーバが同じ計算機内にある場合)?
- ... 分散的文脈では (異なる計算機内にある場合)?

まとめ

概念: 同期的 メッセージ交換 - チャンネル
非同期的 メッセージ交換 - ポート
- send と receive / selective receive
待合わせ 双方向通信 - エントリ
- call と accept ... reply

モデル: チャンネル : ラベルの付け替え, 選択 と ガード
ポート : メッセージ待ち行列, 選択 と ガード
エントリ : ポート と チャンネル

実践: 分散計算 (個別記憶)
スレッド と モニタ (共有記憶)

Course Outline

- ◆ Processes and Threads
- ◆ Concurrent Execution
- ◆ Shared Objects & Interference
- ◆ Monitors & Condition Synchronization
- ◆ Deadlock
- ◆ Safety and Liveness Properties
- ◆ Model-based Design

Concepts
Models
Practice

- ◆ Dynamic systems
- ◆ Concurrent Software Architectures
- ◆ **Message Passing**
- ◆ Timed Systems

課題

3章から10章にある課題をやること

- ◆ 次週までに少なくとも1題を解いて, それを次回発表する.
- ◆ 少なくとも3題を解いて, それをレポートとして提出する.
 - 締切: 2月5日(火)18:00
 - 提出方法: 玉井に直接提出するか学内便で送付