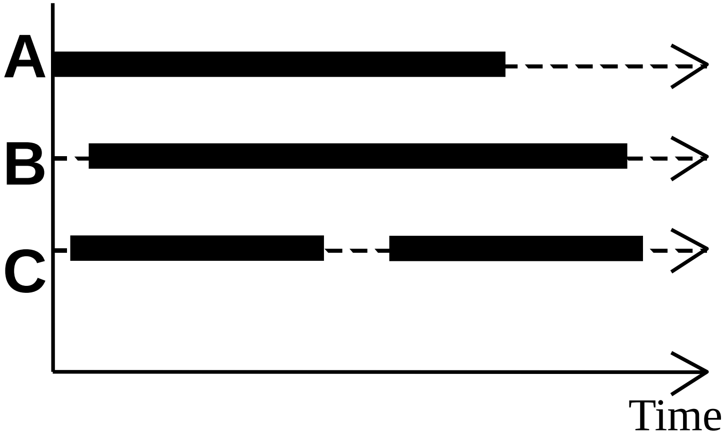
Lecture 4

- Administration
  - Piazza – update with name

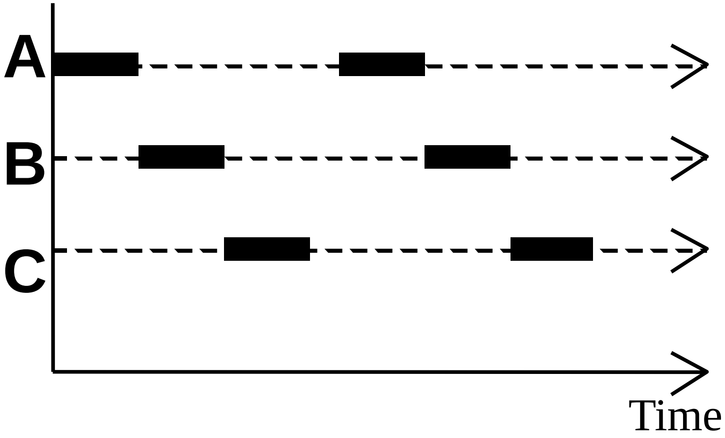
- Concurrency as Interleaving, synchronization

Parallelism vs. Concurrency

Parallelism



Concurrency



Both **concurrency** and **parallelism** require controlled access to *shared resources*.

We use the terms parallel and concurrent interchangeably (and generally do not distinguish between real and pseudo-concurrent execution).

Also, creating software independent of the physical setup, makes us capable of deploying it on *any* platform!

Properties of the System

- A *property* is a predicate evaluated over an execution(s) of the concurrent system (a trace(s))
- Example property: There is no trace where the operators foot is pressing on the brake and the car is accelerating.
- Not a property: the average speed of the car is X KMs/sec.

Traces, Prefixes, and Predicates

- Sequence: $a_1a_2a_3a_4a_5$,…
- Prefixes of this sequence: a_1, a_1a_2, $a_1a_2a_3$, etc.
- A *predicate* is a formula evaluated to a boolean value (true or false)
  - E.g., $x > 10$

Two Types of properties

- ❑ Two types of formal properties in asynchronous computation:
    - m Safety Properties
    - m Liveness Properties

Safety Properties

- A *safety* property is of the form *nothing bad happens* (that is, all states are safe)
- Examples:
  - Action "a" happens before action "b" in all traces.
  - Every delivered message was previously sent
  - My FedEx package was not lost
- A safety property has to hold for all executions (FOR ALL)

Liveness Properties

- A liveness property is of the form *something good happens eventually*
  - An interesting state is eventually achieved
- Examples:
  - There are traces where "a" occurs N times, for all N.
  - My vehicle will accelerate.
  - Eventually I will be served in the restaurant.
- A liveness property may be false for some prefix but can be fixed. (THERE EXISTS?)

So Far

- (x-> P)
- (x -> P | y -> Q)
- (when B x -> P | y -> Q)
- (P || Q)
- Shortcuts for substitution

One more FSP instruction

- if B then P else Q

- If condition B is true then behave like P else behave like Q. (without the else it is equivalent to STOP)

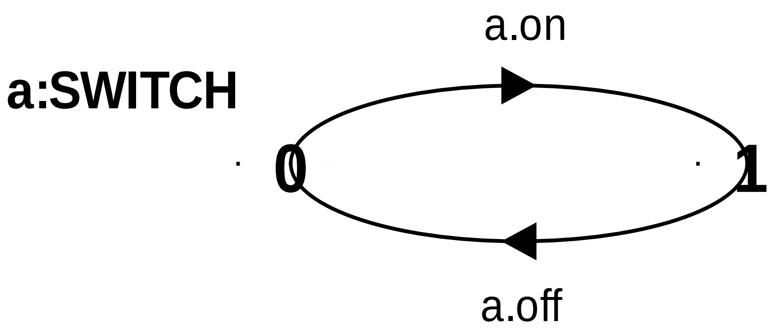
More NAME substitution

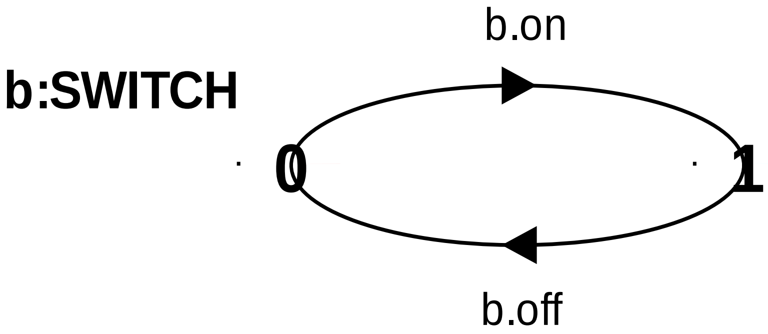
process instances and labeling

a:P prefixes each action label in the alphabet of P with a.

Two **instances** of a switch process:

```
SWITCH =  (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```





An array of **instances** of the switch process:

```
||SWITCHES(N=3) = (forall[i:1..N] s[i]:SWITCH).
||SWITCHES(N=3) = (s[i:1..N]:SWITCH).
```

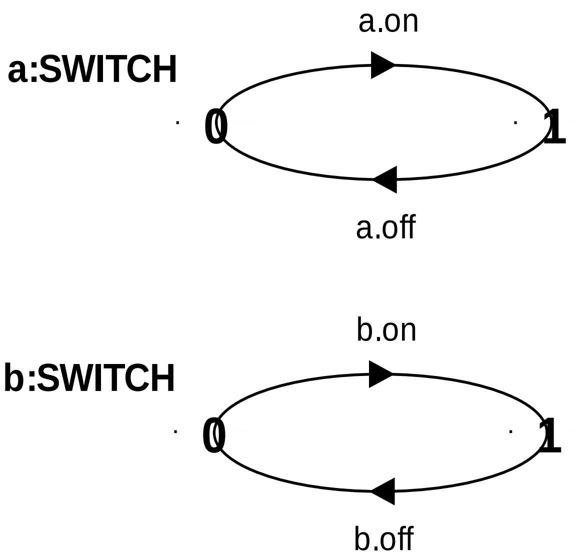
process labelling

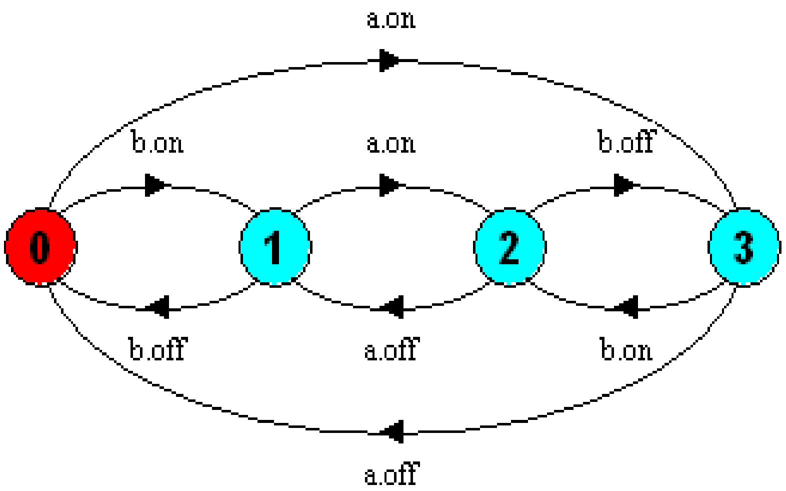
a:P prefixes each action label in the alphabet of P with a.

Two **instances** of a switch process:

LTS? (a:SWITCH)

```
SWITCH =  (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```





process labeling by a set of prefix labels

{a1,..,ax}::P replaces every action label n in the alphabet of P with the labels a1.n,…,ax.n. Further, every transition (n->X) in the definition of P is replaced with the transitions ({a1.n,…,ax.n} ->X).

Process prefixing is useful for modelling **shared** resources:

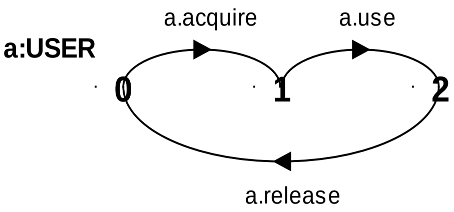
```
RESOURCE = (acquire->release->RESOURCE).
USER = (acquire->use->release->USER).

||RESOURCE_SHARE = (a:USER || b:USER
                           || {a,b}::RESOURCE).
```

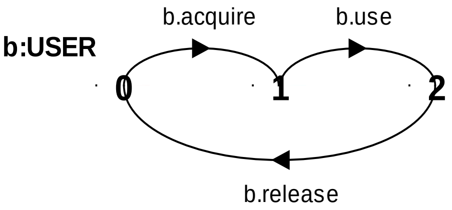
process prefix labels for shared resources

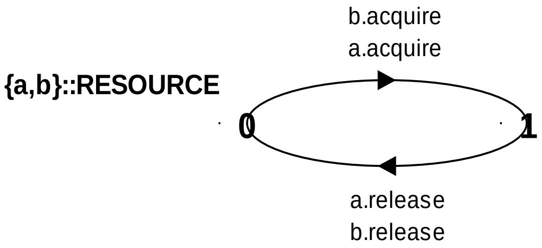
```
RESOURCE = (acquire->release->RESOURCE).
```

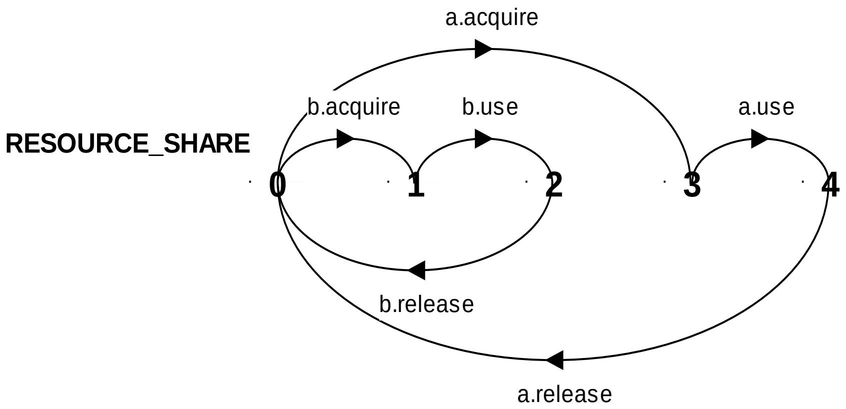
```
USER     = (acquire->use->release->USER).
```

```
||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).
```









How does the model ensure that the user that acquires the resource is the one to release it?

action relabeling

Relabeling functions are applied to processes to change the names of action labels. The general form of the relabeling function is:

$/\{newlabel_1/oldlabel_1,\ldots\ newlabel_n/oldlabel_n\}$.

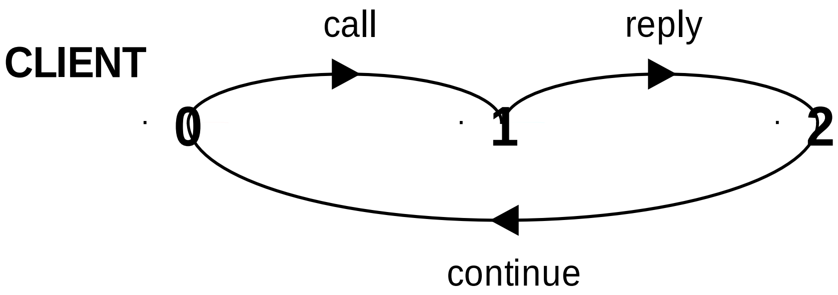
Relabeling to ensure that composed processes synchronize on particular actions.

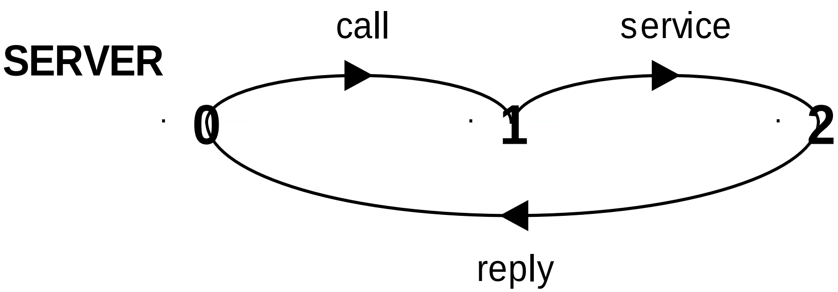
```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
```

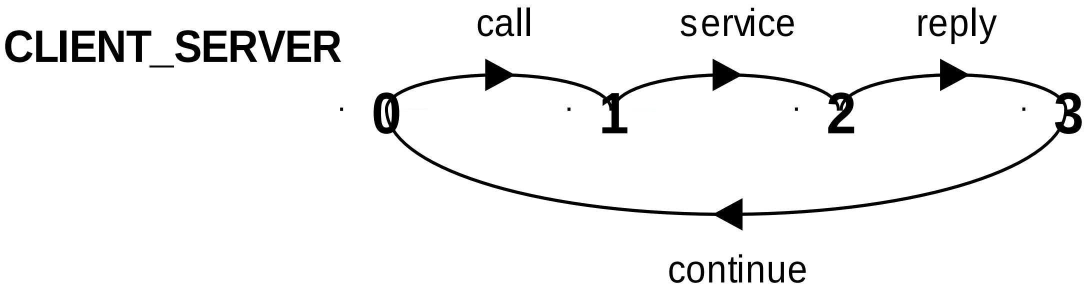
Note that both *newlabel* and *oldlabel* can be sets of labels.

action relabeling

```
||CLIENT_SERVER = (CLIENT || SERVER)
                 /{call/request, reply/wait}.
```







action relabeling - prefix labels

An alternative formulation of the client server system is described below using qualified or prefixed labels:

```
SERVERv2 = (accept.request
            ->service->accept.reply->SERVERv2).
CLIENTv2 = (call.request
            ->call.reply->continue->CLIENTv2).


||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)
                    /{call/accept}.
```

action hiding - abstraction to reduce complexity

When applied to a process P, the hiding operator \{a1..ax} removes the action names a1..ax from the alphabet of P and makes these concealed actions "silent". These silent actions are labeled tau. Silent actions in different processes are not shared.

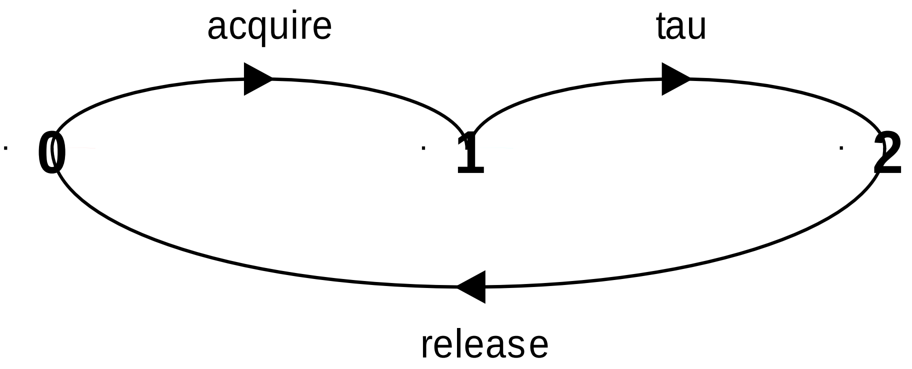
Sometimes it is more convenient to specify the set of labels to be exposed....

When applied to a process P, the interface operator @{a1..ax} hides all actions in the alphabet of P not labeled in the set a1..ax.

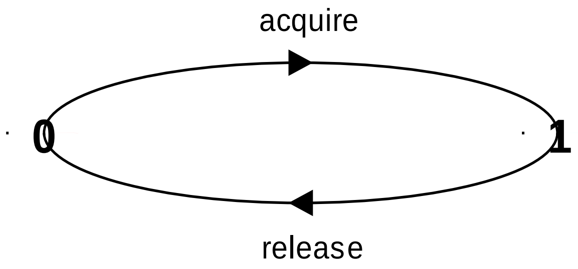
action hiding

The following definitions are equivalent:

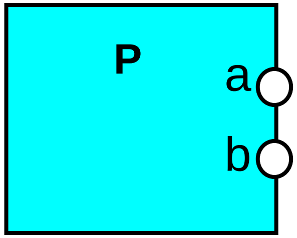
```
USER = (acquire->use->release->USER)
       \{use}.

USER = (acquire->use->release->USER)
          @{acquire,release}.
```



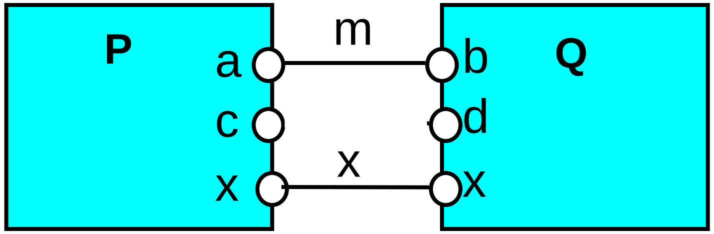
Minimization removes hidden **tau** actions to produce an LTS with equivalent observable behavior.



structure diagrams

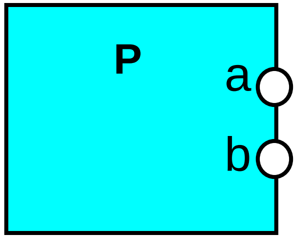


Process P with alphabet {a,b}.

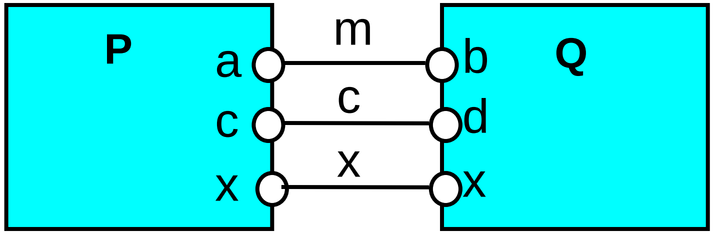


Parallel Composition
(P||Q) / {m/a,m/b }

structure diagrams

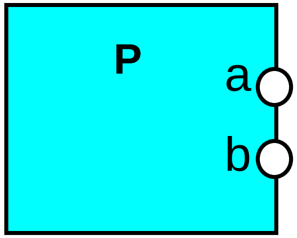


Process P with alphabet {a,b}.

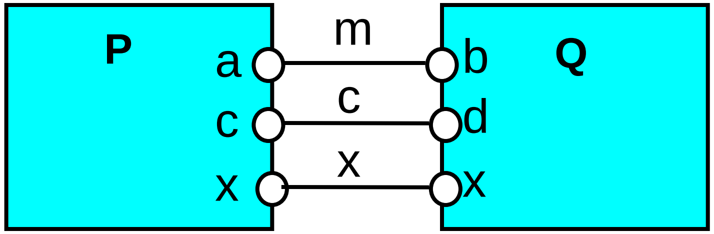


Parallel Composition
(P||Q) / {m/a,m/b,c/d}

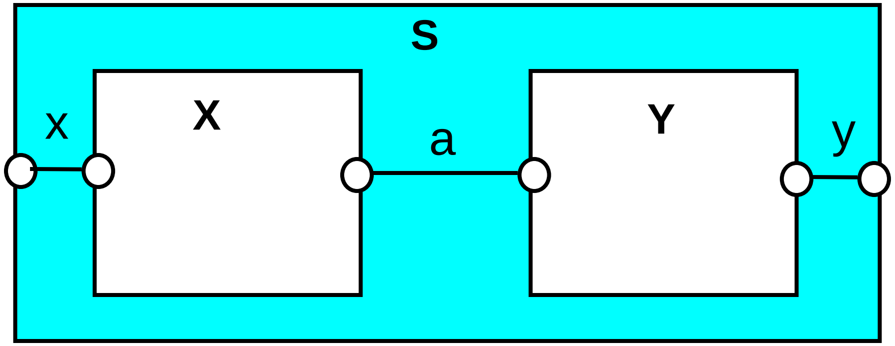
structure diagrams



Process P with
alphabet {a,b}.



Parallel Composition
(P||Q) / {m/a,m/b,c/d}

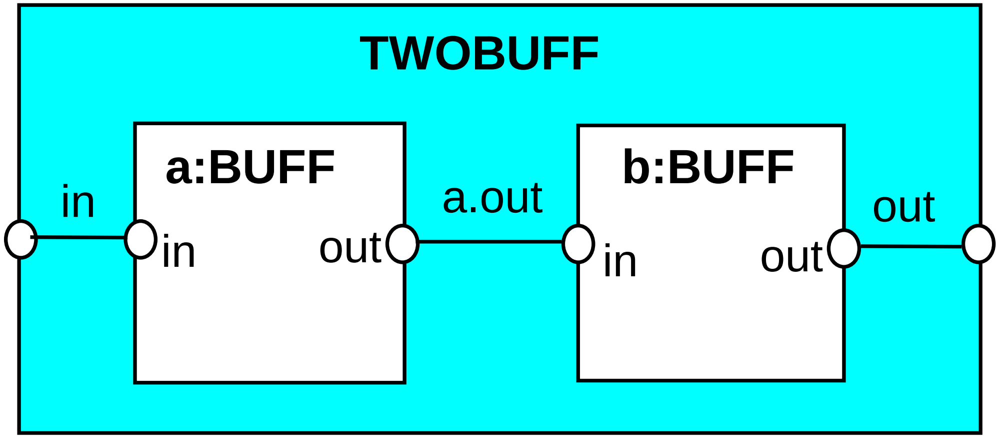


Composite process
||S = (X||Y) @ {x,y}

structure diagrams

We use structure diagrams to capture the structure of a model expressed by the static combinators: *parallel composition*, *relabeling* and *hiding*.

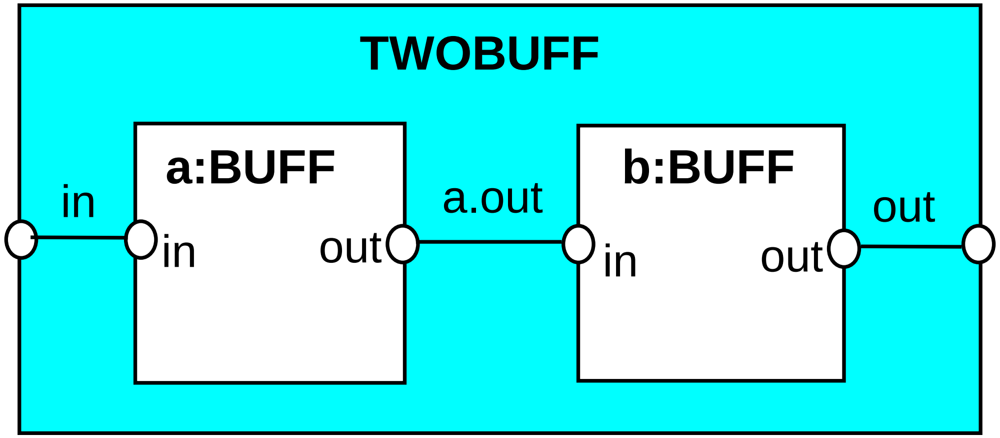
```
range T = 0..3
BUFF = (in[i:T]->out[i]->BUFF).

||TWOBUF = ?
```



structure diagrams

```
range T = 0..3
BUFF = (in[i:T]->out[i]->BUFF).
```

We use structure diagrams to capture the structure of a model expressed by the static combinators: *parallel composition*, *relabeling* and *hiding*.

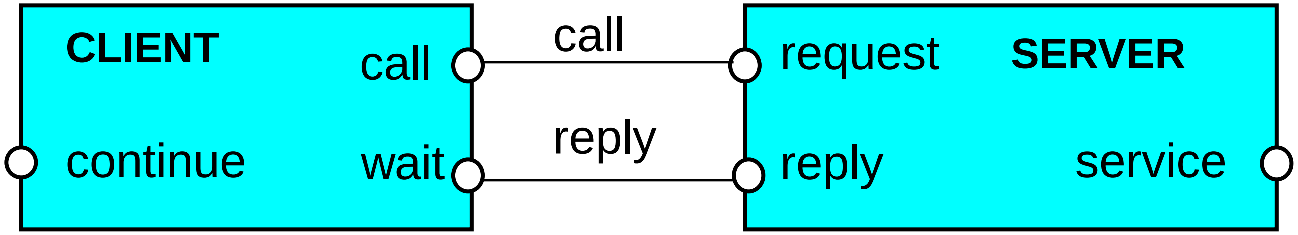


```
||TWOBUF =      (a:BUFF || b:BUFF)
                   /{in/a.in, a.out/b.in, out/b.out}
                      @{in,out}.
```

structure diagrams

```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).

||CLIENT_SERVER = (CLIENT||SERVER)
                             /{reply/wait,
                               call/request}.
```

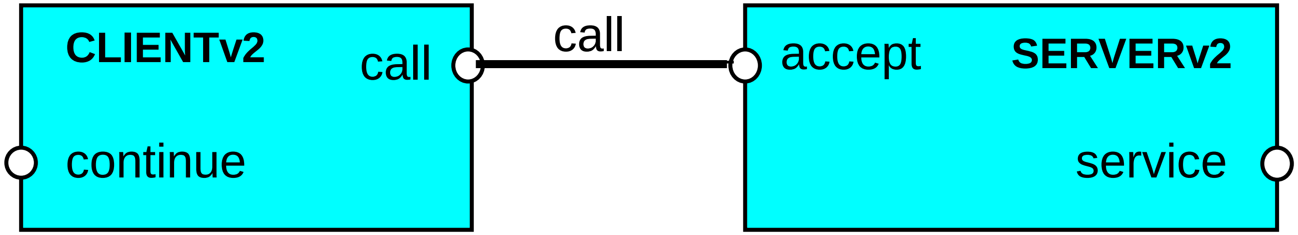
Structure diagram for **CLIENT_SERVER** ?



structure diagrams

```
SERVERv2 = (accept.request
            ->service->accept.reply->SERVERv2).
CLIENTv2 = (call.request
            ->call.reply->continue->CLIENTv2).

||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)
                    /{call/accept}.
```

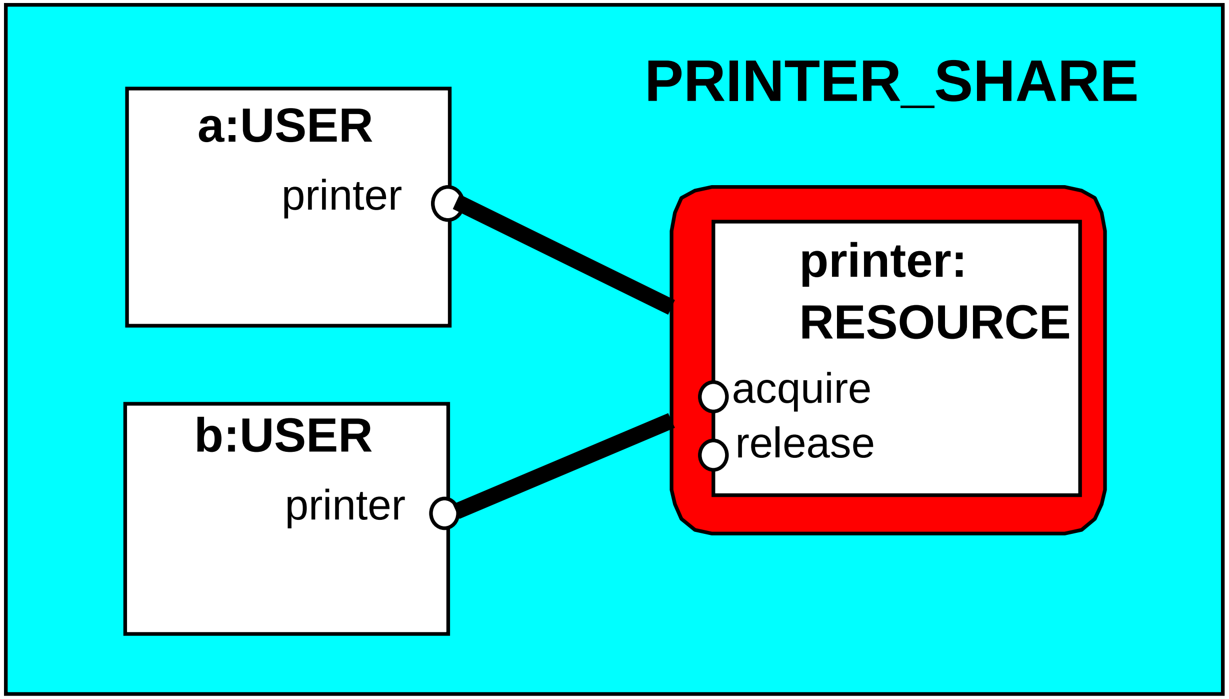
Structure diagram for **CLIENT_SERVERv2** ?



structure diagrams - resource sharing

```
RESOURCE = (acquire->release->RESOURCE).
USER     = (printer.acquire->use->printer.release->USER).
```

```
||PRINTER_SHARE =
   (a:USER || b:USER || {a,b}::printer:RESOURCE).
```


PRINTER_SHARE
a:USER
printer
b:USER
printer
printer:
RESOURCE
acquire
release