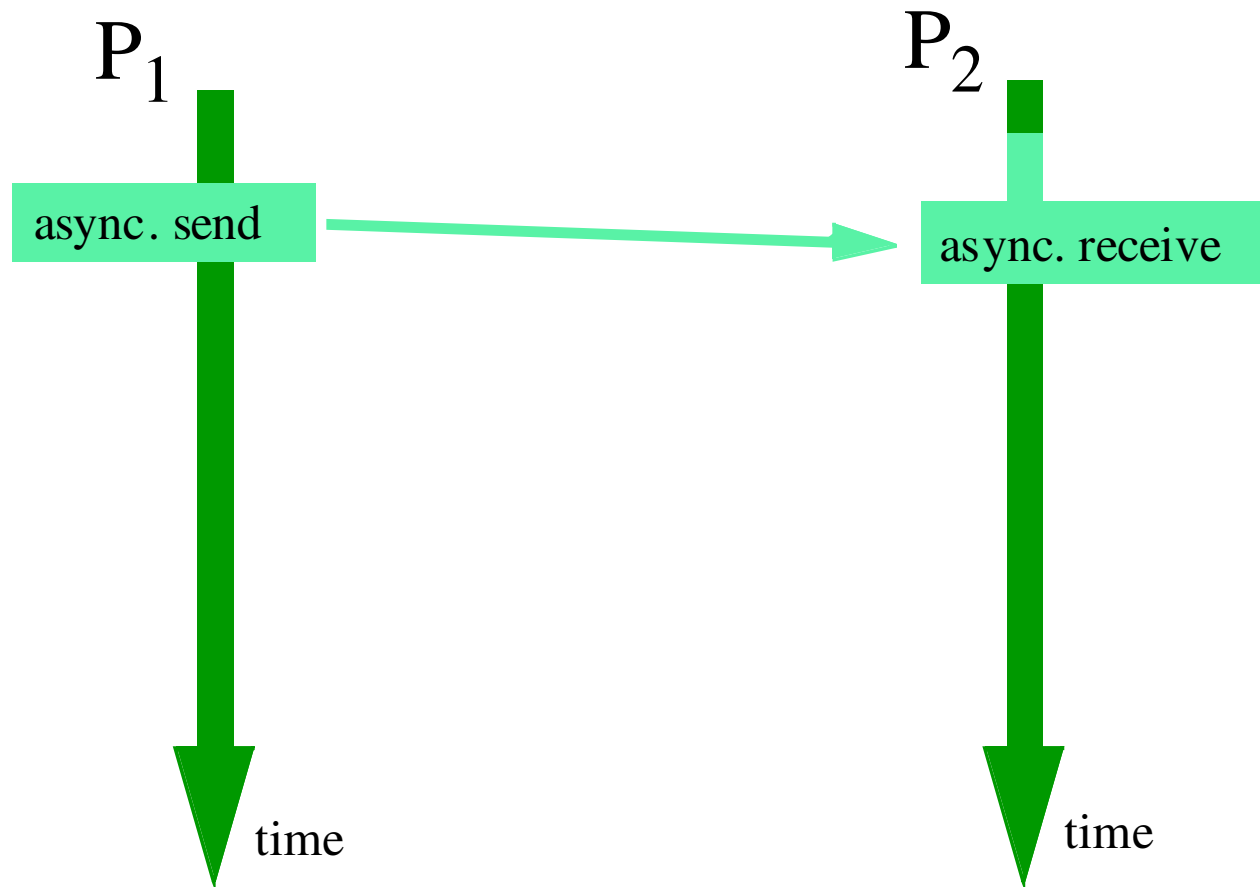


# *Synchronization#3:* *Message Based*

# *Message-based Synchronization*

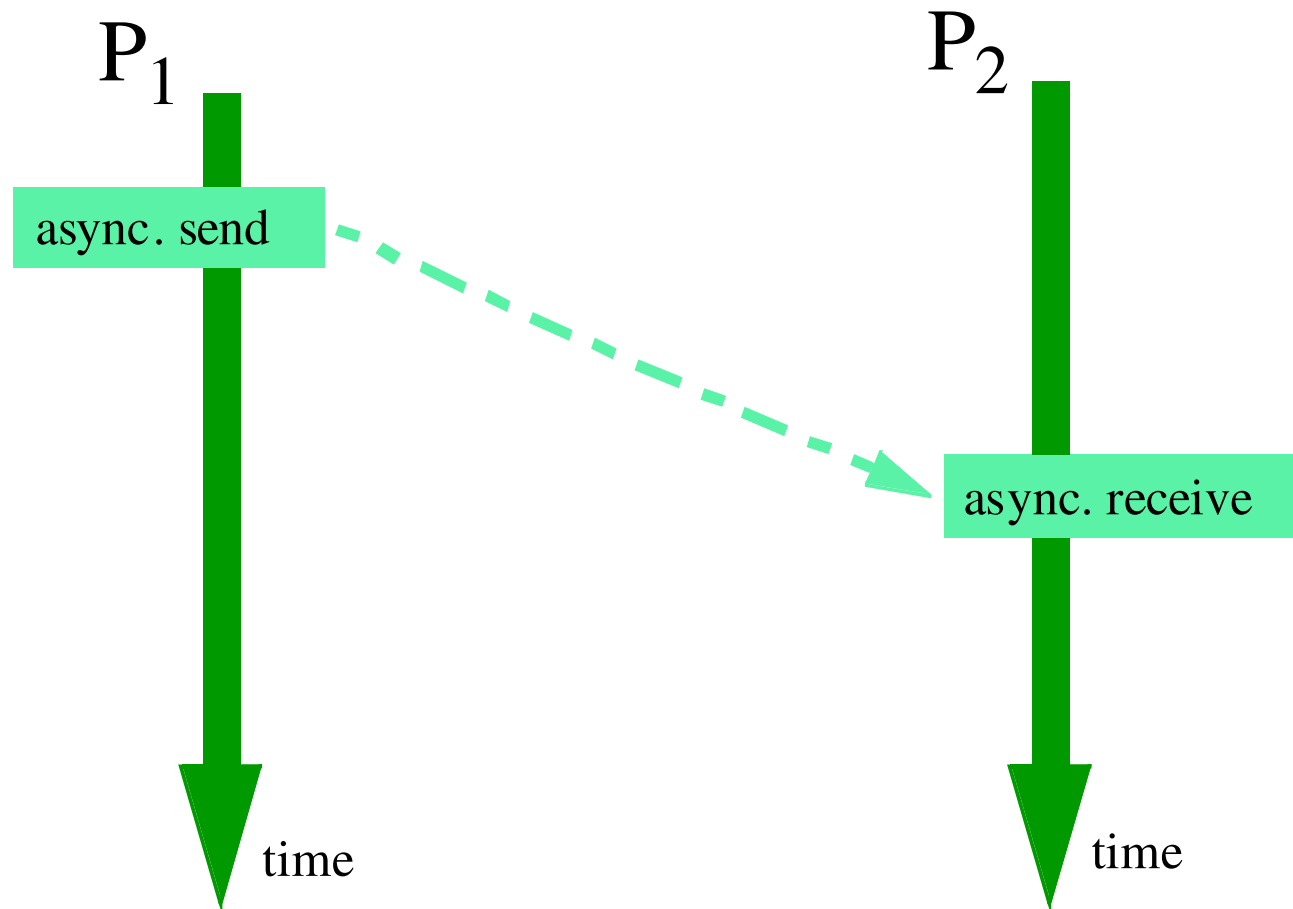
- Synchronization model
  - Asynchronous
  - Synchronous
  - Remote invocation
- Addressing (name space)
  - direct communication
  - mail-box communication
- Message structure
  - arbitrary
  - restricted to 'basic' types
  - restricted to un-typed communications

# Asynchronous Messages



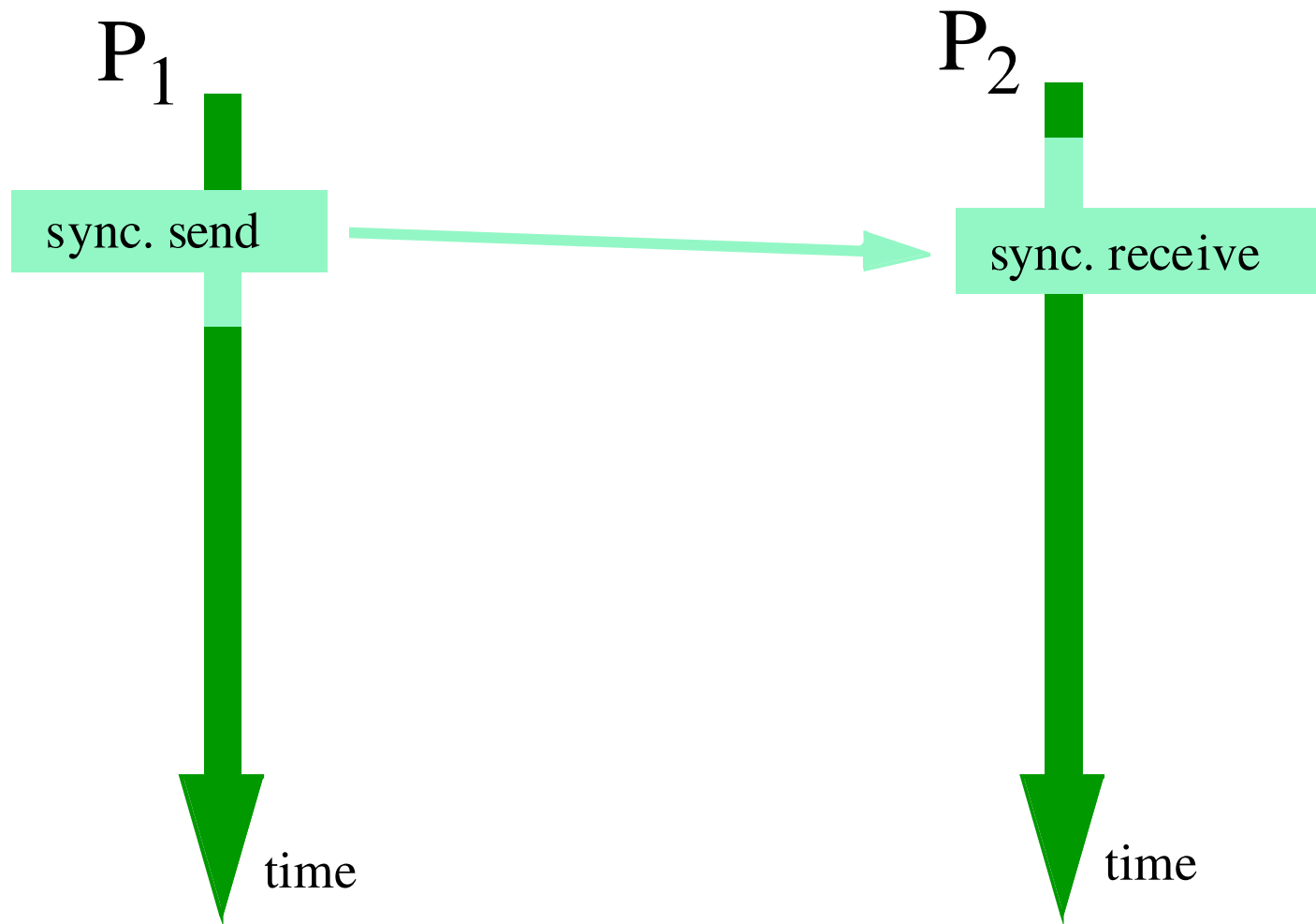
- If there is a listener:
  - send the message directly

# Asynchronous Messages



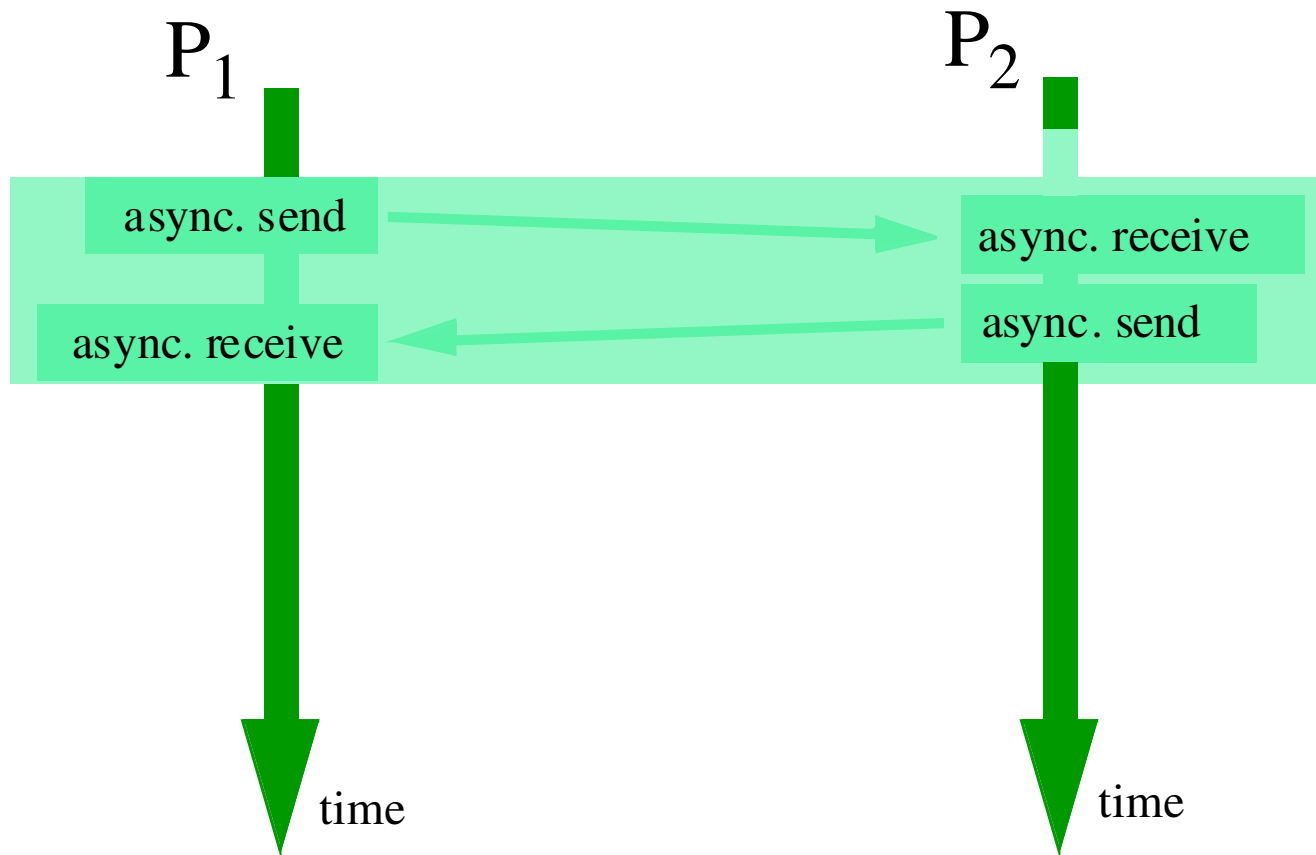
- If the receiver becomes available at a later stage:
  - the message needs to be buffered

# Synchronous Messages



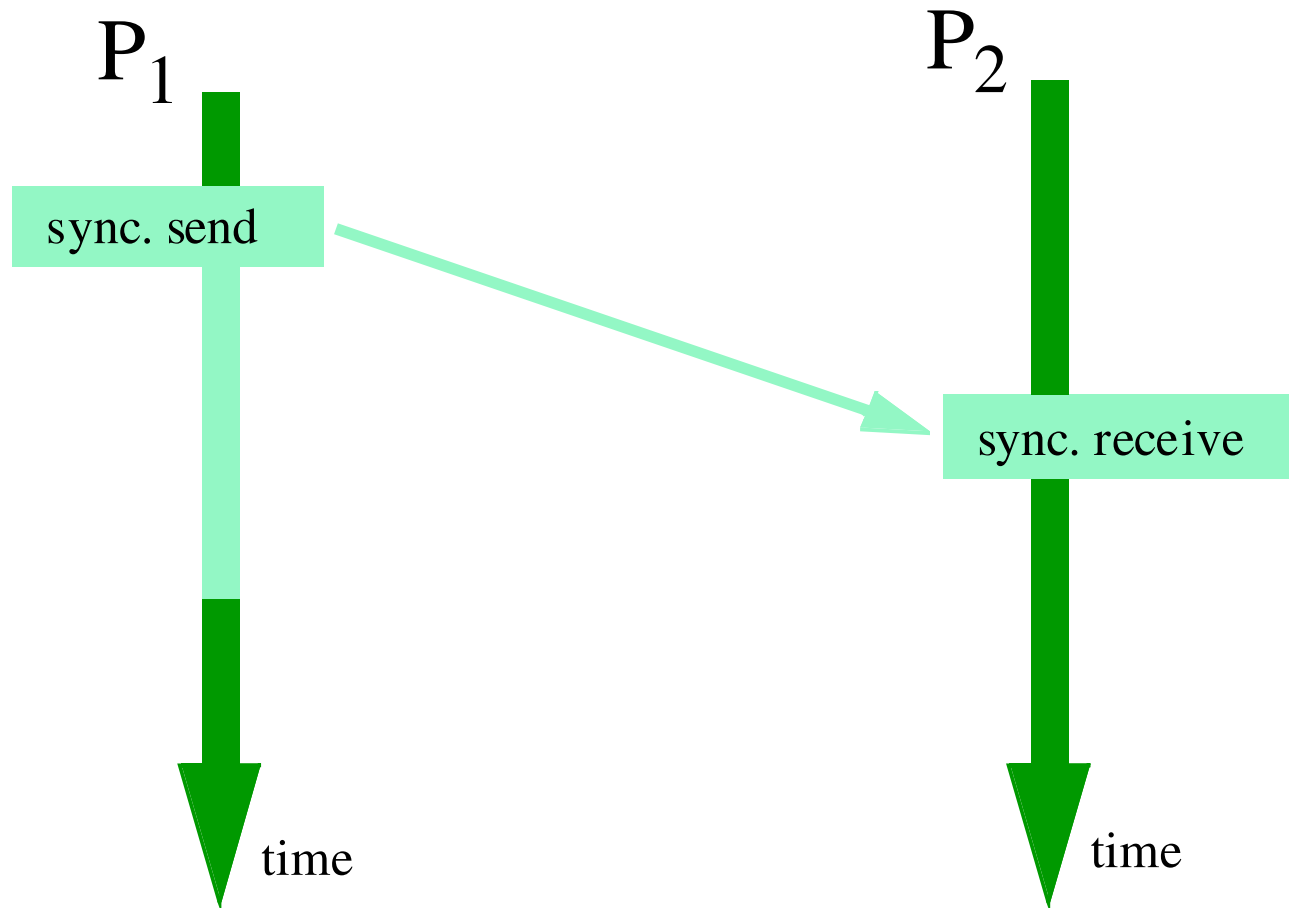
- Delay the receiver:
  - until the message becomes available

# Synchronous Messages



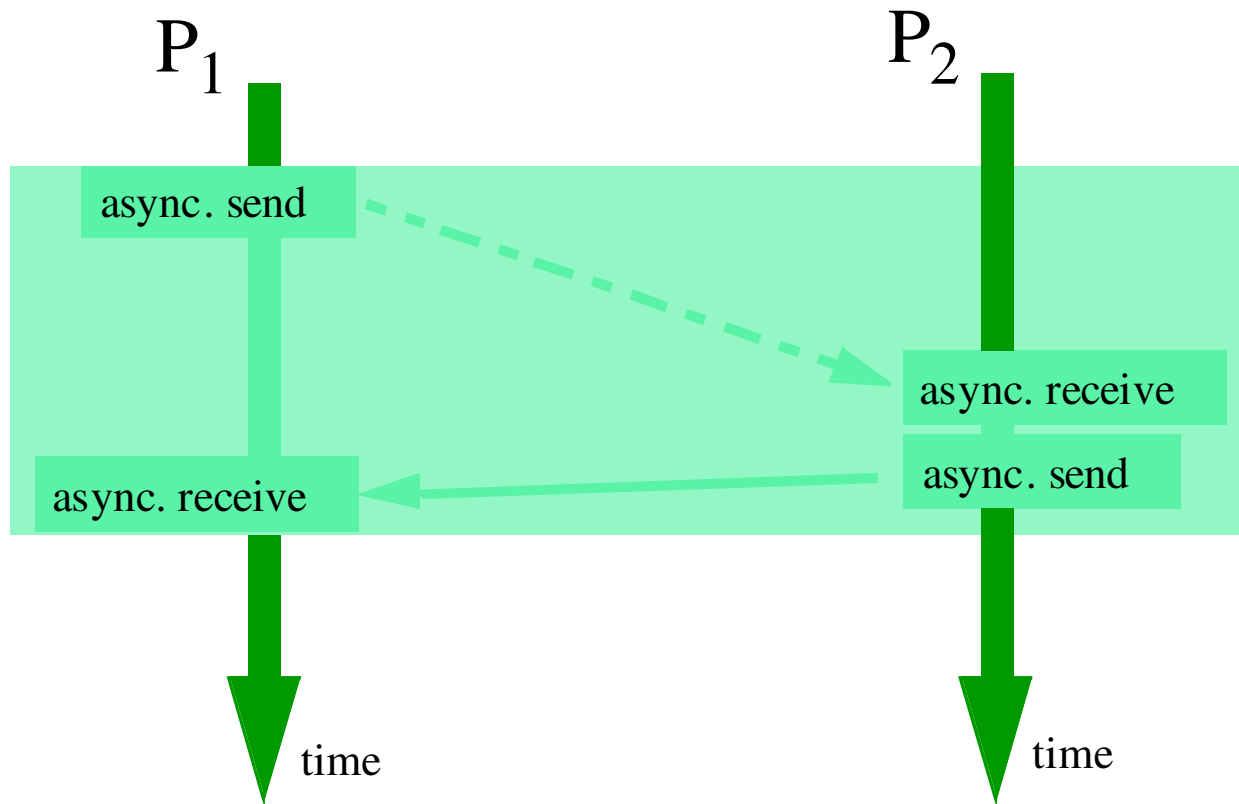
- Delay the receiver:
  - until the message becomes available
- Simulated by asynchronous messages:
  - two asynchronous messages required

# Synchronous Messages



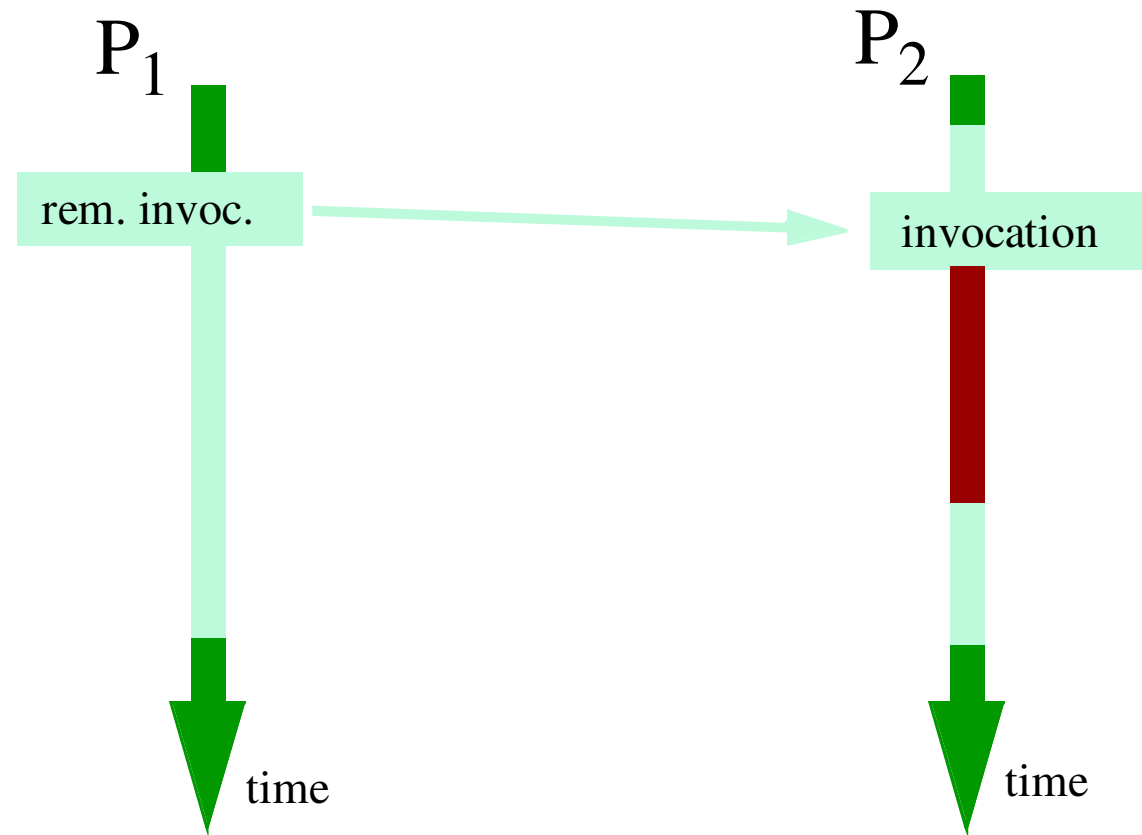
- Delay the sender until:
  - a receiver is available
  - a receiver got the message

# Synchronous Messages



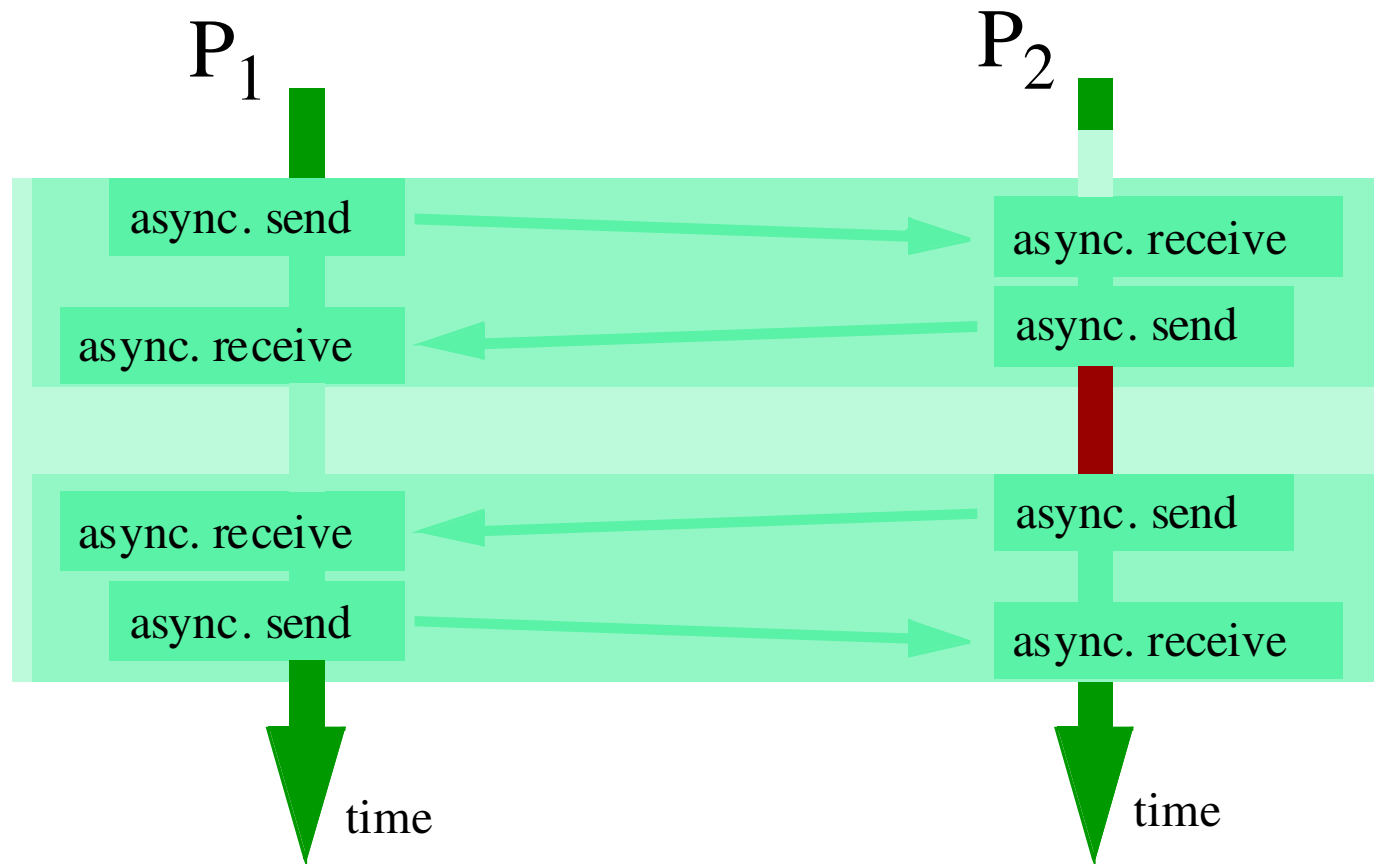
- Delay the sender until:
  - a receiver is available
  - a receiver got the message
- Simulated by asynchronous messages and receiver becomes available at a later stage:
  - message needs to be buffered

# Remote Invocation



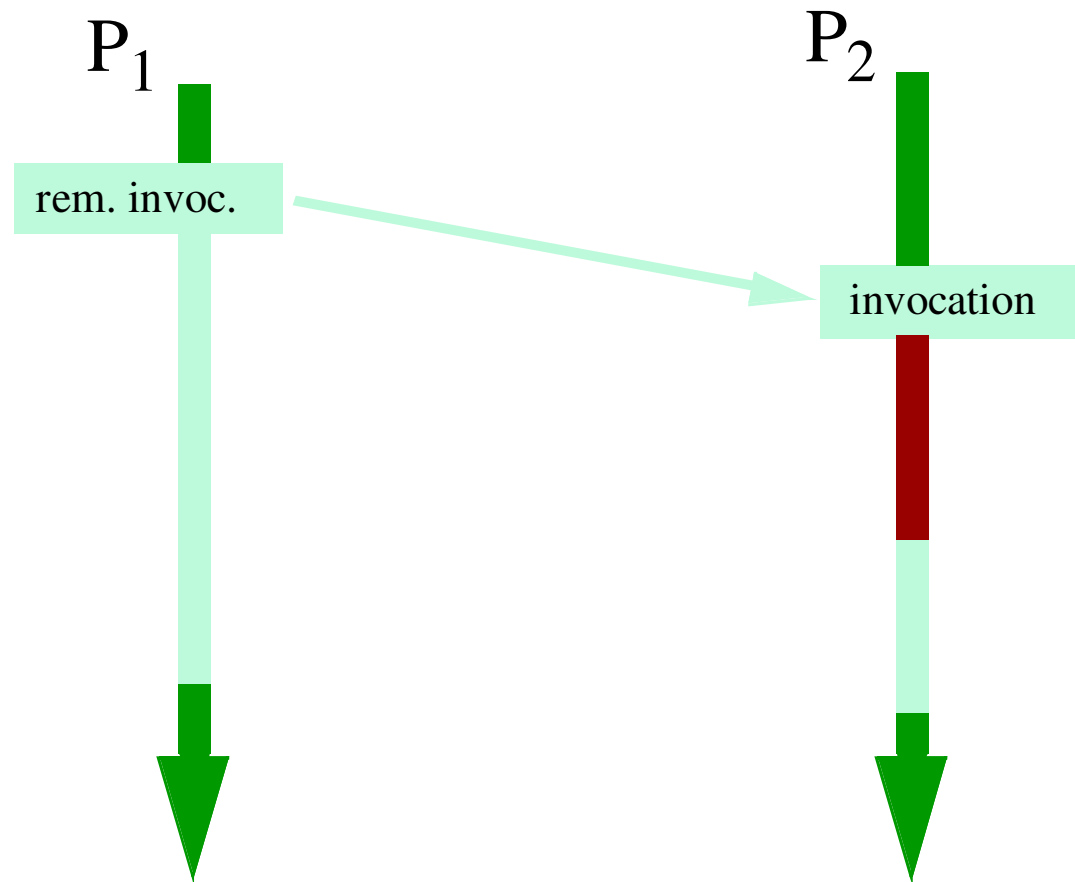
- Delay the receiver, until:
  - an invocation is available
  - a receiver executed an addressed routine

# Remote Invocation



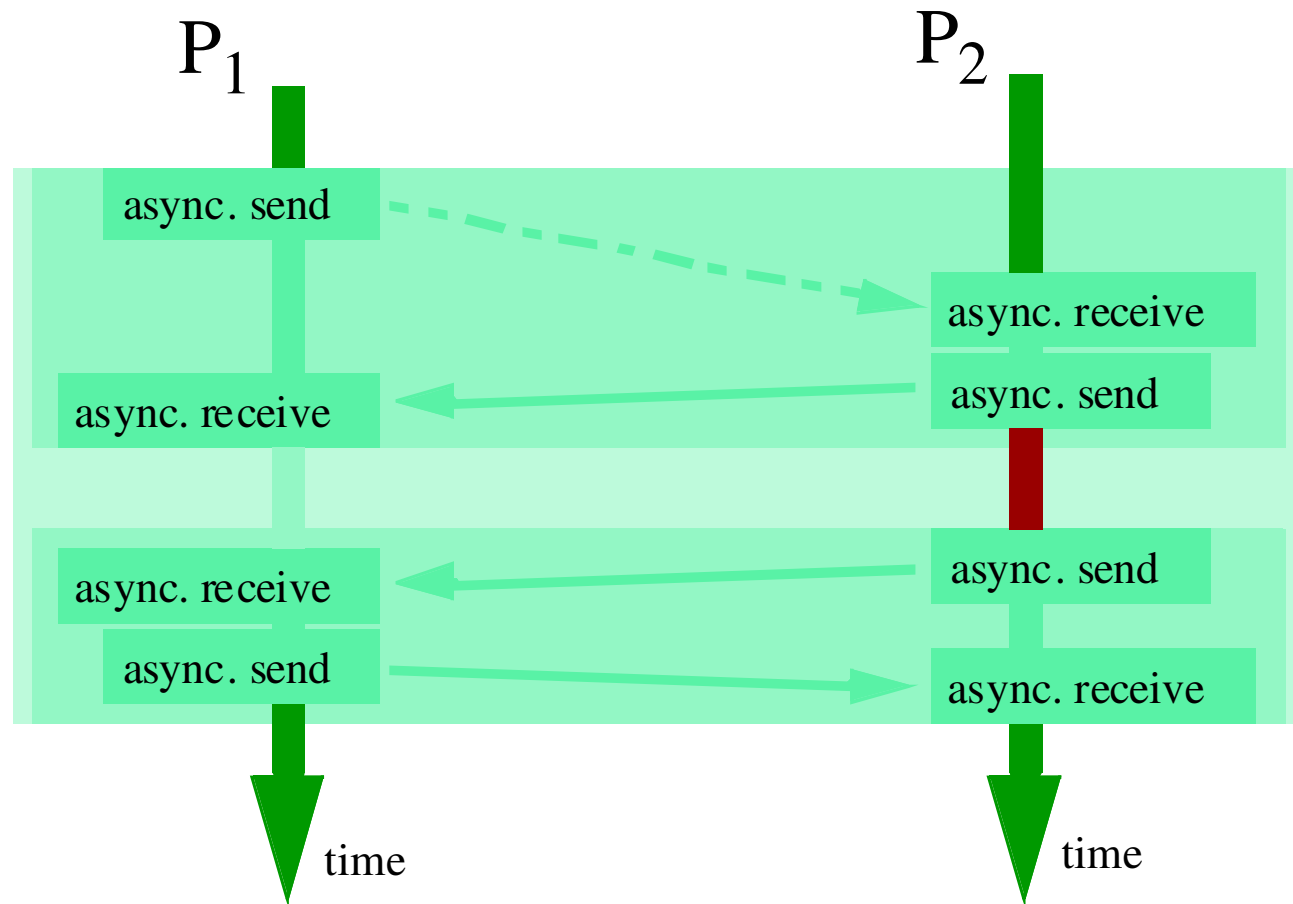
- Delay the receiver, until:
  - an invocation is available
  - a receiver executed an addressed routine
- Simulated by asynchronous messages:
  - four messages are required

# Remote Invocation



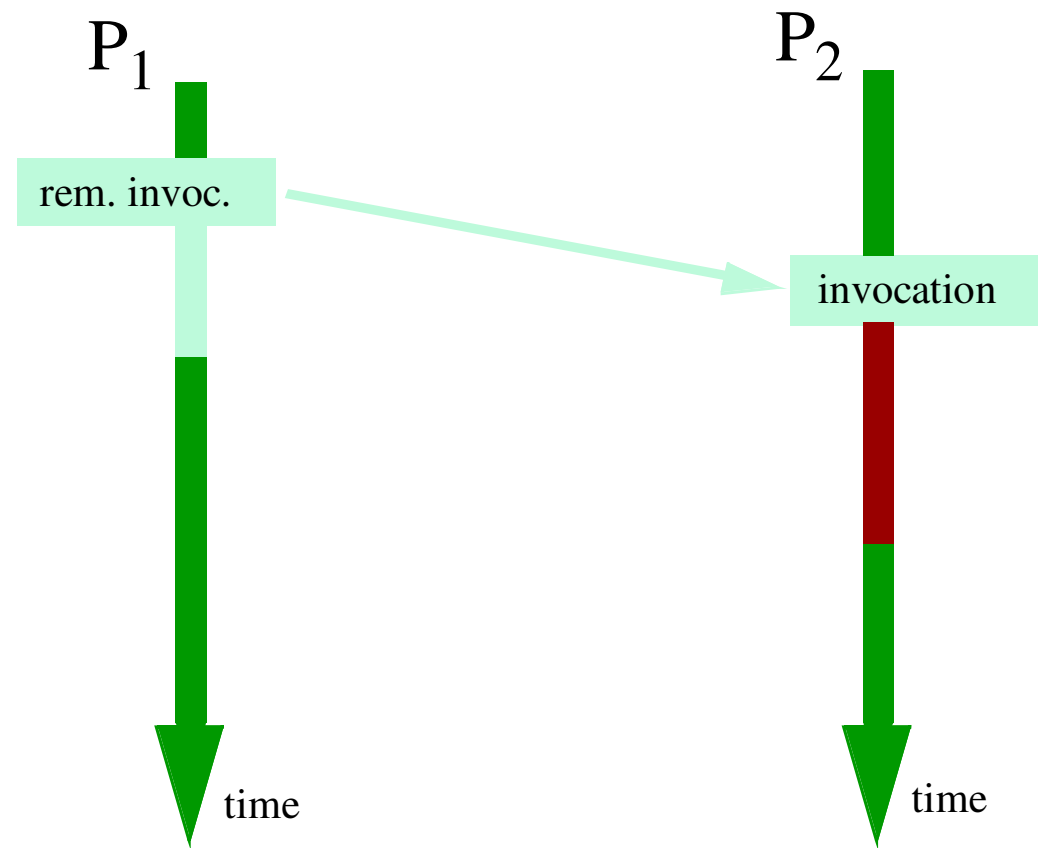
- Delay the sender, until:
  - a receiver becomes available
  - a receiver got the message
  - a receiver executed an addressed routine

# Remote invocation



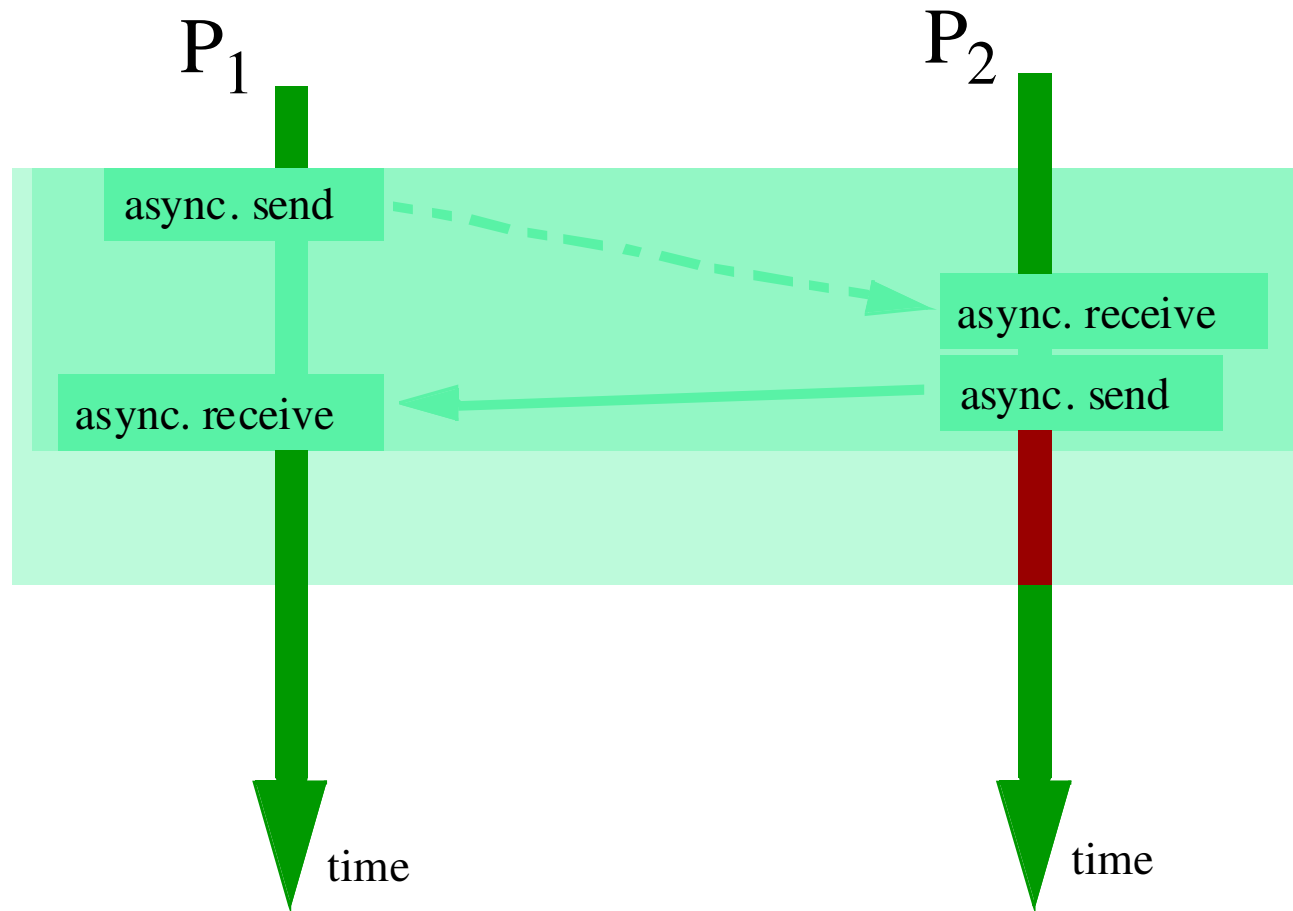
- Delay the sender, until:
  - a receiver becomes available
  - a receiver got the message
  - a receiver executed an addressed routine
- Simulated by asynchronous messages:
  - four messages are required
  - message buffering required

# Asynchronous Remote Invocation



- Delay the sender, until:
  - a receiver becomes available
  - a receiver got the message

# Asynchronous Remote Invocation



- Delay the sender, until:
  - a receiver becomes available
  - a receiver got the message
- Simulated by asynchronous messages:
  - two messages are required

# *Synchronous vs. Asynchronous*

- **Purpose**
  - **synchronization**: synchronous messages / remote invocations
  - **in-time delivery**: asynchronous messages / asynchronous remote invocations
- ‘Real’ synchronous message passing in distributed systems requires hardware support.
- Asynchronous message passing requires the usage of (infinite?) buffers.
- **Emulation**
  - Synchronous communications are emulated by a combination of asynchronous messages in some systems.
  - Asynchronous communications can be emulated in synchronized message passing systems by introducing ‘buffer-tasks’ (decoupling sender and receiver as well as allowing for broadcasts).

# Addressing (*name space*)

- **Direct vs. indirect:**

```
send      <message> to    <process-name>
wait for  <message> from  <process-name>
send      <message> to    <mailbox>
wait for  <message> from  <mailbox>
```

- **Asymmetrical addressing:**

```
send      <message> to ...
wait for  <message>
```

– Client-server paradigm

# Addressing (*name space*)

Communication medium:

<b><i>Connections</i></b>	<b><i>Functionality</i></b>
one-to-one	buffer, queue, synchronization
one-to-many	multicast
one-to-all	broadcast
many-to-one	local server, synchronization
all-to-one	general server, synchronization
many-to-many	general network- or bus-system

# *Message structure*

- Machine dependent representations need to be taken care of in a distributed environment.
- Communication system is often outside the typed language environment.
  - Most communication systems are handling streams (packets) of a basic element type only.
- Conversion routines for data-structures other than the basic element type are supplied ...
  - manually (POSIX, 'C/C++', Java)
  - semi-automatic (CORBA)
  - automatic and are typed-persistent (Ada95, CHILL, Occam2)

# *Message Structure (Ada95)*

```
package Ada.Streams is
  pragma Pure (Streams);
  type Root_Stream_Type is abstract tagged limited private;
  type Stream_Element is mod implementation-defined;
  type Stream_Element_Offset is range implementation-defined;
  subtype Stream_Element_Count is
    Stream_Element_Offset range 0..Stream_Element_Offset'Last;
  type Stream_Element_Array is
    array (Stream_Element_Offset range <>) of Stream_Element;
  procedure Read (...) is abstract;
  procedure Write (...) is abstract;
private
  ... -- not specified by the language
end Ada.Streams;
```

# *Message Structure (Ada95)*

## **Reading and writing values of any type to a stream:**

```
procedure S'Write(Stream : access
    Ada.Streams.Root_Stream_Type'Class; Item : in T);
procedure S'Class'Write(Stream : access
    Ada.Streams.Root_Stream_Type'Class; Item : in T'Class);
procedure S'Read(Stream : access
    Ada.Streams.Root_Stream_Type'Class; Item : out T);
procedure S'Class'Read(Stream : access
    Ada.Streams.Root_Stream_Type'Class; Item : out T'Class)
```

## **Reading and writing values, bounds and discriminants of any type to a stream:**

```
procedure S'Output(Stream : access
    Ada.Streams.Root_Stream_Type'Class; Item : in T);
function S'Input(Stream : access
    Ada.Streams.Root_Stream_Type'Class) return T;
```

# Practical Message-Passing Systems

POSIX:	<p>“message queues”:</p> <ul style="list-style-type: none"><li>– <b>ordered indirect [asymmetrical   symmetrical] asynchronous byte-level many-to-many message passing</b></li></ul>
CHILL	<p>“buffers”, “signals”:</p> <ul style="list-style-type: none"><li>– <b>ordered indirect [asymmetrical   symmetrical] [synchronous   asynchronous] typed [many-to-many   many-to-one] message passing</b></li></ul>
Occam2	<p>“channels”:</p> <ul style="list-style-type: none"><li>– <b>indirect symmetrical synchronous fully-typed one-to-one message passing</b></li></ul>
Ada95:	<p>“(extended) rendezvous”:</p> <ul style="list-style-type: none"><li>– <b>ordered direct asymmetrical [synchronous   asynchronous] fully-typed many-to-one remote invocation</b></li></ul>
Java:	no communication via messages available

# Practical Message-Passing Systems

	<b>Ordered</b>	<b>Symmetrical</b>	<b>Assymetrical</b>	<b>Synchronous</b>	<b>Asynchronous</b>	<b>Direct</b>	<b>Indirect</b>	<b>Contents</b>	<b>One-to-one</b>	<b>Many-to-one</b>	<b>Many-to-many</b>	<b>Method</b>
POSIX	*	*	*		*		*	bytes			*	message passing
CHILL	*	*	*	*	*		*	typed		*	*	message passing
Occam2		*		*			*	fully typed	*			message passing
Ada95	*		*	*	*	*		fully typed		*		remote invocation
Java	no communication via messages available											

# Practical Message-Passing Systems: for strict synchronization purposes

	<b>Ordered</b>	<b>Symmetrical</b>	<b>Assymetrical</b>	<b>Synchronous</b>	<b>Asynchronous</b>	<b>Direct</b>	<b>Indirect</b>	<b>Contents</b>	<b>One-to-one</b>	<b>Many-to-one</b>	<b>Many-to-many</b>	<b>Method</b>
POSIX	*	*	*		*		*	bytes			*	message passing
CHILL	*	*	*	*	*		*	typed		*	*	message passing
Occam2		*		*			*	fully typed	*			message passing
Ada95	*		*	*	*	*		fully typed		*		remote invocation
Java	no communication via messages available											

# Message-Based Synchronization in Occam2

- Communication is ensured by means of a 'channel', which:
  - can be used by one writer and one reader process only
  - is synchronous:

```
CHAN OF INT SensorChannel:
PAR
  INT reading:
  SEQ i = 0 FOR 1000
    SEQ
      -- generate reading
      SensorChannel ! reading
  INT data:
  SEQ i = 0 FOR 1000
    SEQ
      SensorChannel ? data
      -- employ data
```

tasks are synchronized  
at these points

# Message-Based Synchronization in CHILL

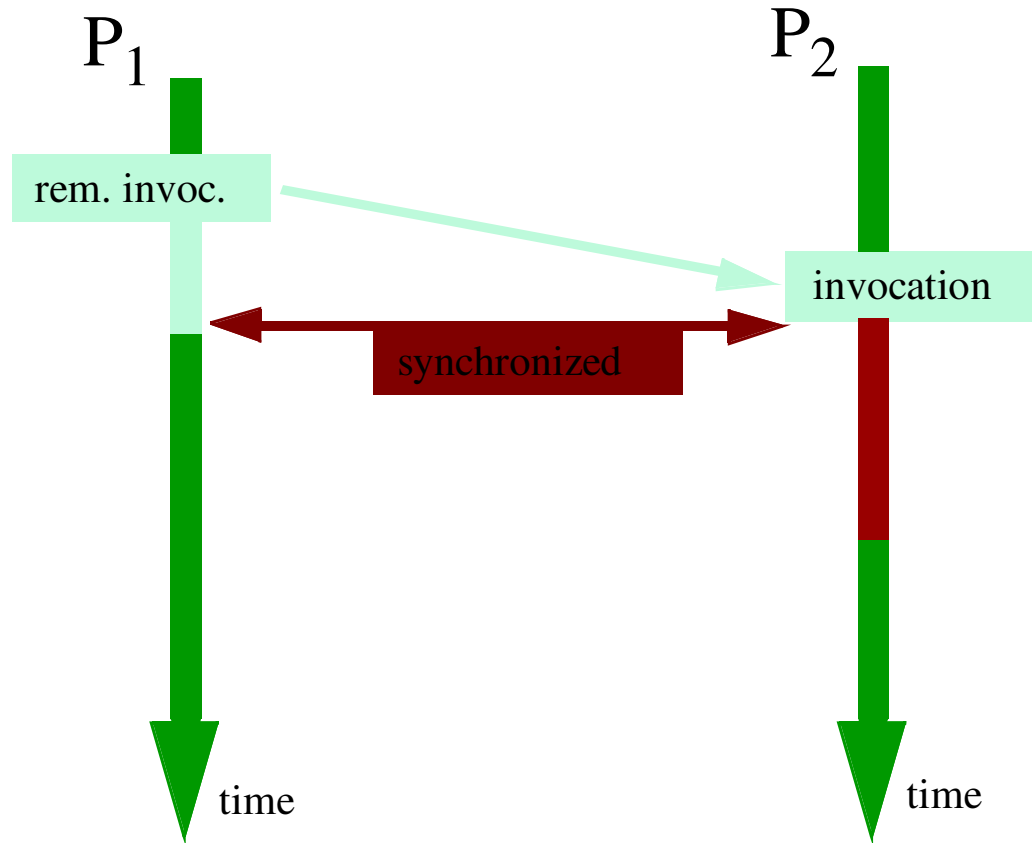
- **CHILL** is the 'CCITT High Level Language',
  - where **CCITT** is the Comité Consultatif International Télégraphique et Téléphonique.
- The CHILL language development was started in 1973 and standardized in 1979.
  - strong support for concurrency, synchronization, and communication (monitors, buffered message passing, synchronous channels)

```
dcl SensorBuffer buffer (32) int;
...
send SensorBuffer (reading); | receive case
    ----- asynchronous -----> (SensorBuffer in data) :
...
    | esac;
signal SensorChannel = (int) to consumertype;
...
send SensorChannel (reading) | receive case
    to consumer ← synchronous → (SensorChannel in data) :
...
    | esac;
```

## *Message-based synchronization in Ada95*

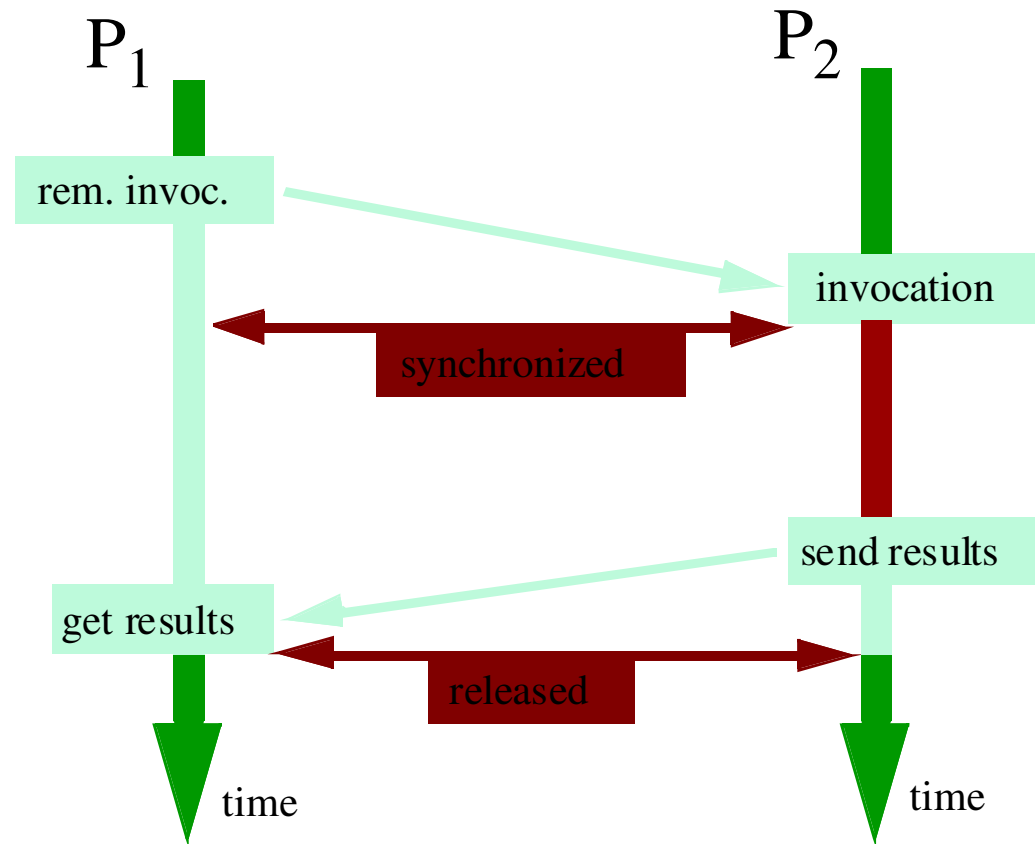
- Ada95 supports remote invocations ((extended rendezvous) in form of:
  - **entry points** in tasks
  - **full set of parameter profiles** supported
- If the local and the remote task are on different architectures, or if an intermediate communication system is employed:
  - parameters incl. bounds and discriminants are ‘tunnelled’ through byte-stream-formats.
- Synchronization:
  - both tasks are synchronized at the beginning of the remote invocation (+ ‘**rendezvous**’)
  - the calling task is blocked until the remote routine is completed (+ ‘**extended rendezvous**’)

# Remote invocation: Rendezvous



- Delay the sender, until:
  - a receiver becomes available
  - a receiver got the message
  - a receiver started an addressed routine

# Remote Invocation: Extended Rendezvous



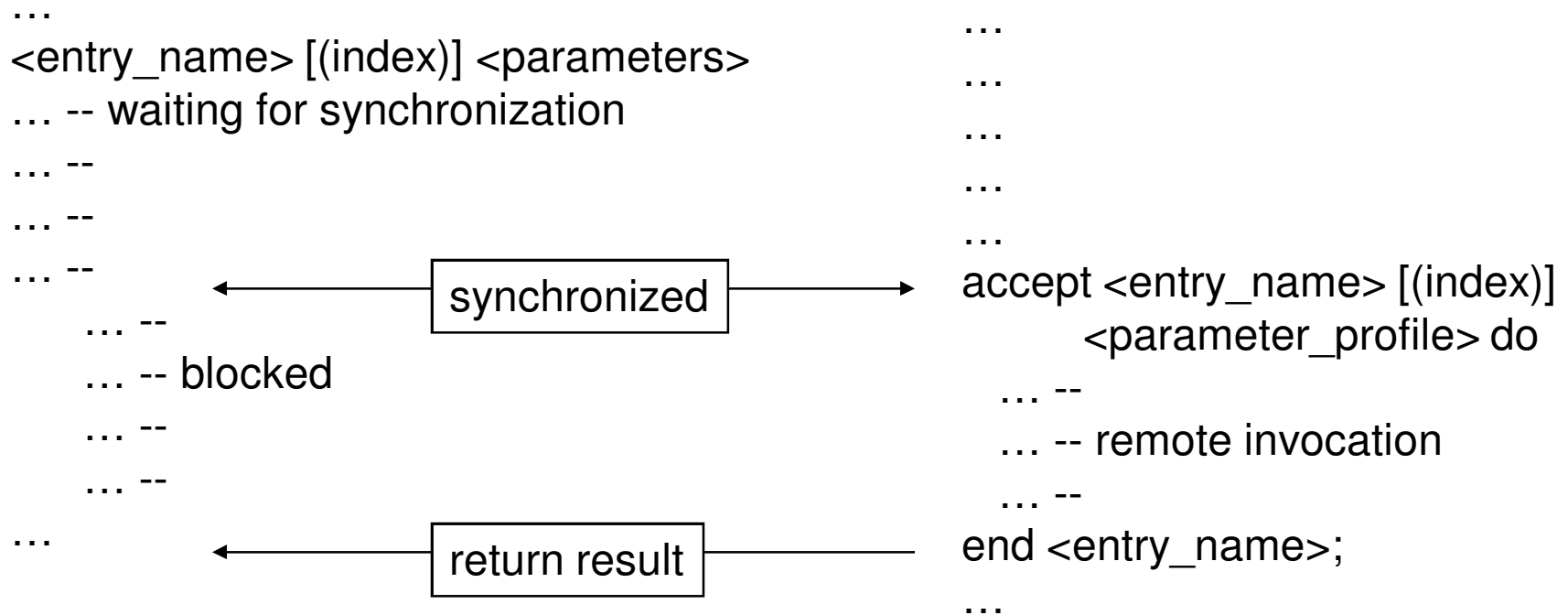
- Delay the sender, until:
  - a receiver becomes available
  - a receiver got the message
  - a receiver executed an addressed routine
  - a receiver passed the results

# Ada95: Rendezvous

```
...
<entry_name> [(index)] <parameters> ...
.. -- waiting for synchronization ..
.. -- ..
.. -- ..
.. -- ← synchronized → accept <entry_name> [(index)]
.. .. <parameter_profile>;
.. ..
.. ..
.. ..
.. ..
.. ..
```



# Ada95: Extended Rendezvous





# Task Entries

- In contrast to protected-object-entries, task-entries *can* call other blocking operations.
- Accept statements can be nested (but need to be different).
  - helpful e.g. to synchronize more than two tasks.
- Accept statements can have a dedicated exception handler (like any other code-block).
- Exceptions, which are not handled during the rendezvous phase are propagated to *all* involved tasks.
- Parameters cannot be direct ‘access’ parameters, but can be access-types.
- ‘count on task-entries is defined, but is only accessible from inside the tasks owning the entry.
- **Entry families** (arrays of entries) are supported.
- **Private entries** (accessible for internal tasks) are supported.

# *Synchronization*

- **Shared memory based synchronization**
  - Flags, condition variables, semaphores, ...  
... conditional critical regions, monitors, protected objects.
  - Guard evaluation times, nested monitor calls, deadlocks, ...  
... simultaneous reading, queue management.
  - Synchronization and object orientation, blocking operations and re-queuing.
- **Message based synchronization**
  - Synchronization models
  - Addressing modes
  - Message structures
  - Examples