Distributed Systems

DAT520 - Spring 2026

Chapter 1 - Introduction

Prof. Hein Meling

University of Stavanger

Programming Abstractions

Organizing Principle

Encapsulate distributed interactions within

- **Abstractions** with *well-defined interfaces*
- To help us:
  - Reason about the **correctness** of distributed applications

Well-known Programming Abstractions

- Sequential programming abstractions
  - Set, Map, List
- Concurrent programming abstractions
  - Thread, Semaphore, Monitor, Locks

Distributed Programming Abstractions

This Course

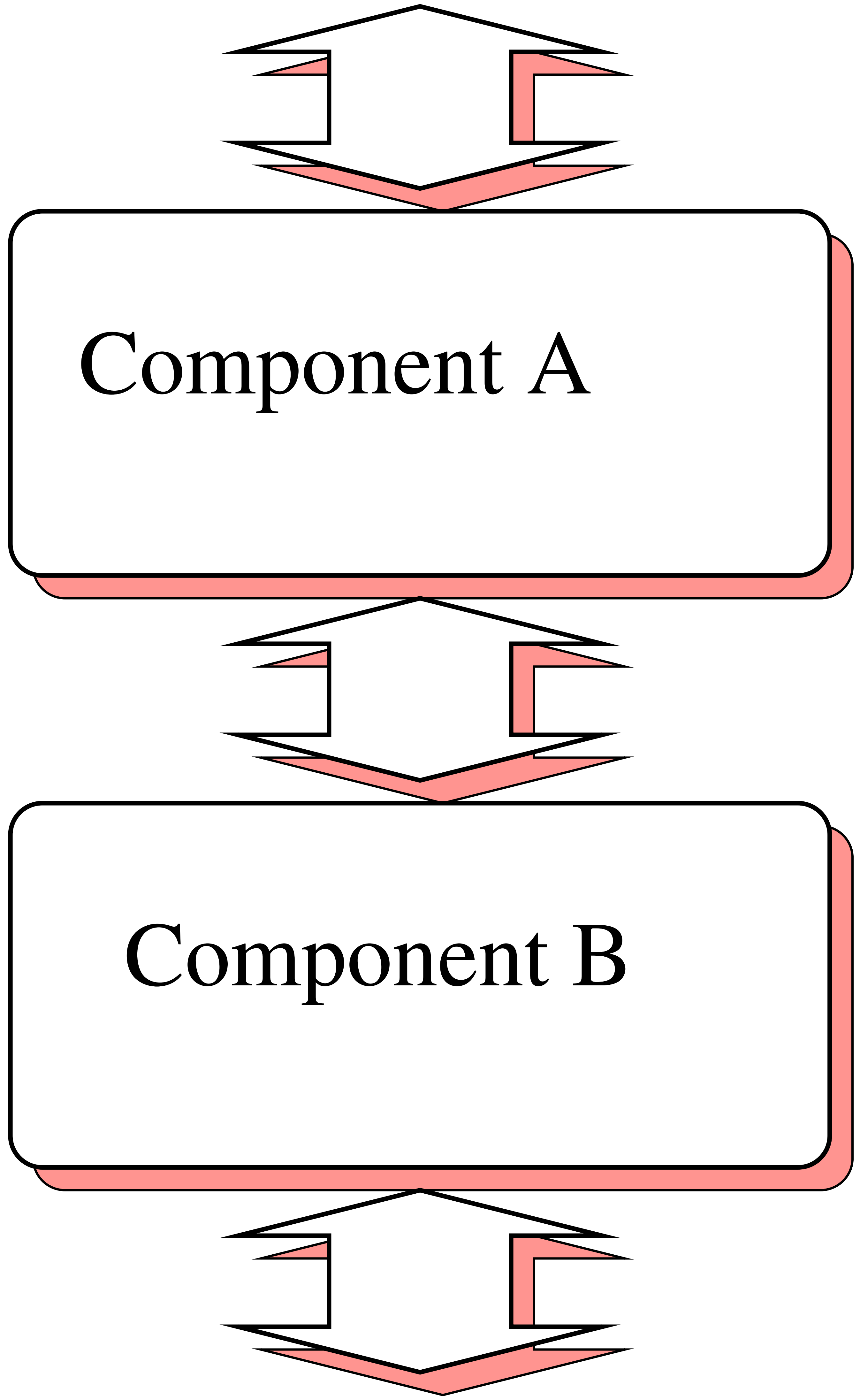
- Reliable broadcast
- Causal order broadcast
- Total order broadcast
- Consensus
- Shared memory
- Atomic commit
- Leader election
- Group membership

In-process Interaction

1.4 Software Components

Composition Model

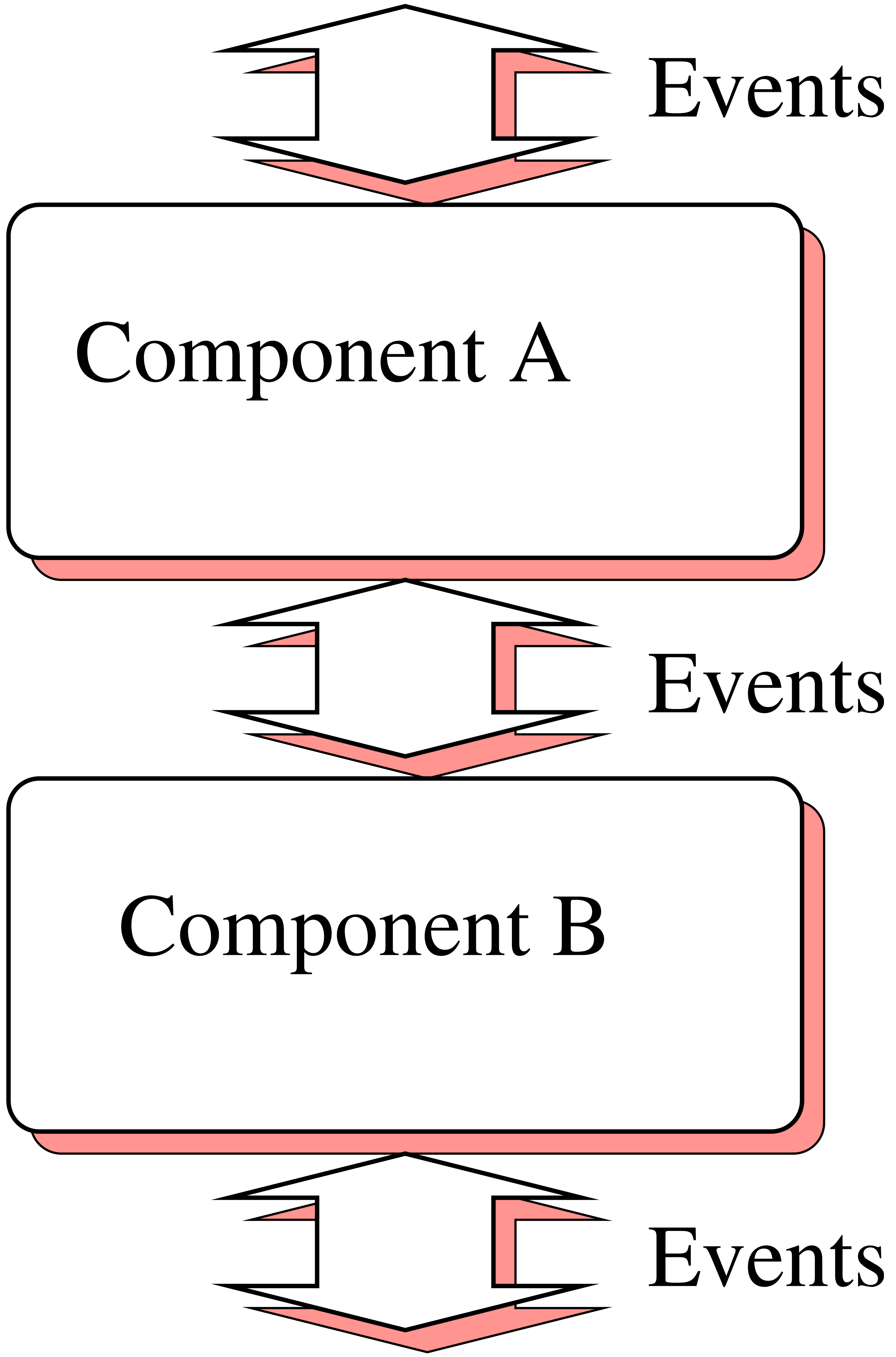
- The Program (or **Process**):
    - Composed of a finite set of *protocol modules*
- Organized into a stack or graph
    - Layer = Module = Component



1.4 Software Components

Reactive Computing Model

- Modules (*same process*)
  - Interact by **exchanging events**
- Algorithm (implemented by *a module*)
  - Described as a set of **event handlers**

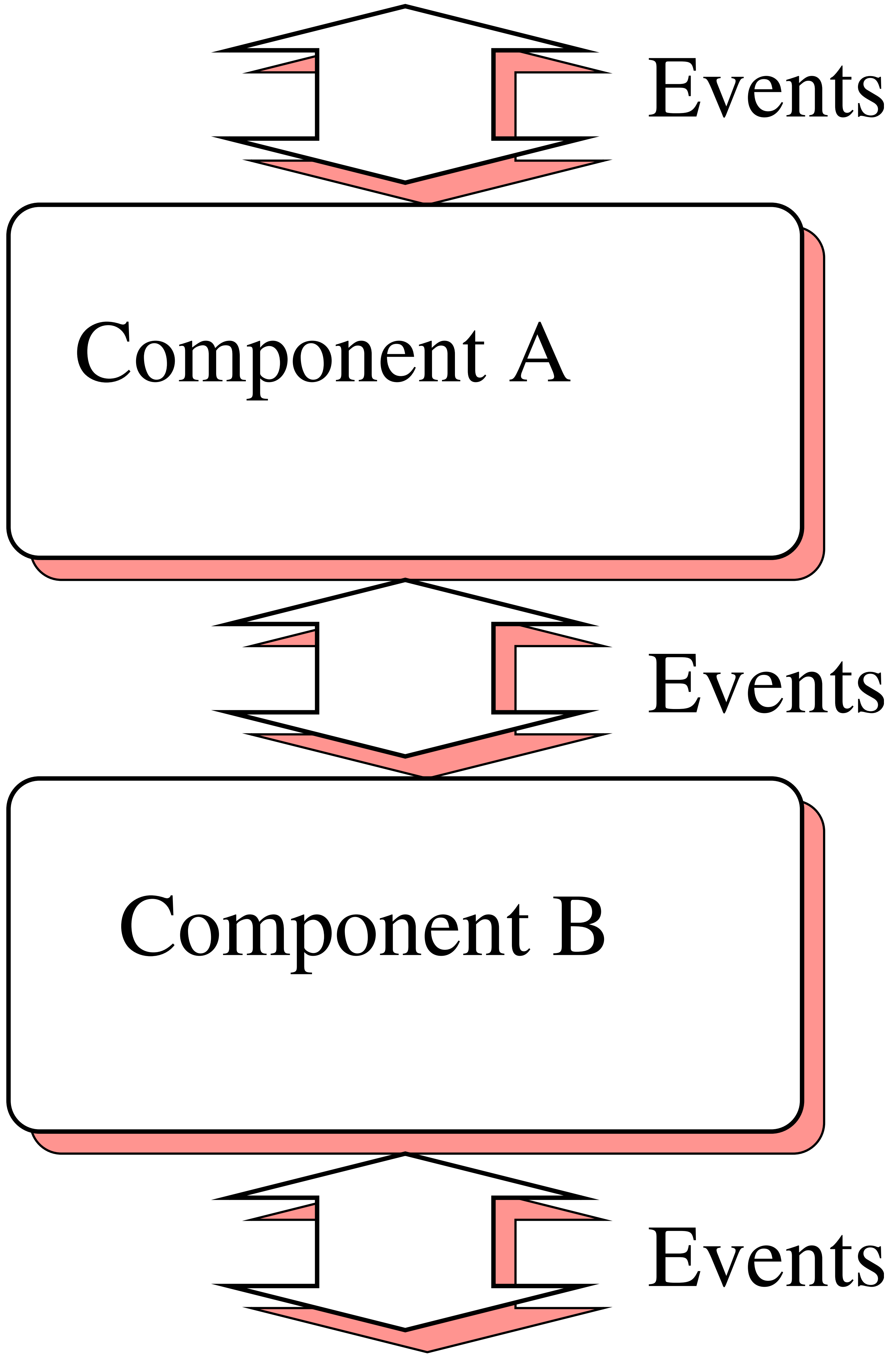


1.4 Software Components

Reactive Computing Model

- Modules (*same process*)
  - Interact by **exchanging events**
- Algorithm (implemented by *a module*)
  - Described as a set of **event handlers**

$$
\begin{aligned}
&\textbf{upon event} \ \langle\ co_1, Event_1 \mid att_1^1, att_1^2, \ldots \rangle \ \textbf{do} \\
&\qquad \text{do something;} \\
&\qquad \textbf{trigger} \ \langle\ co_2, Event_2 \mid att_2^1, att_2^2, \ldots \rangle;
\end{aligned}
$$

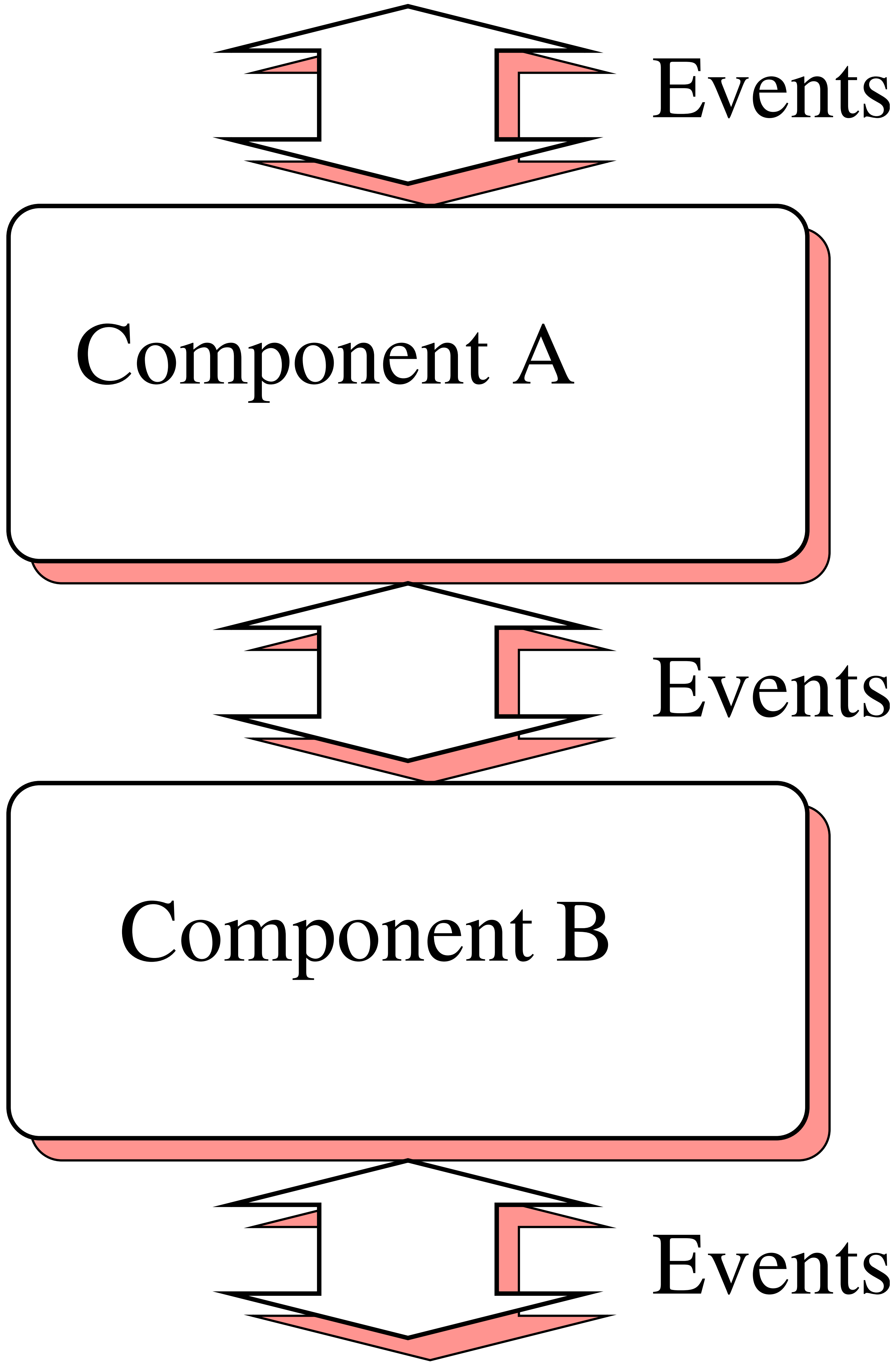


1.4 Software Components

Event Handlers

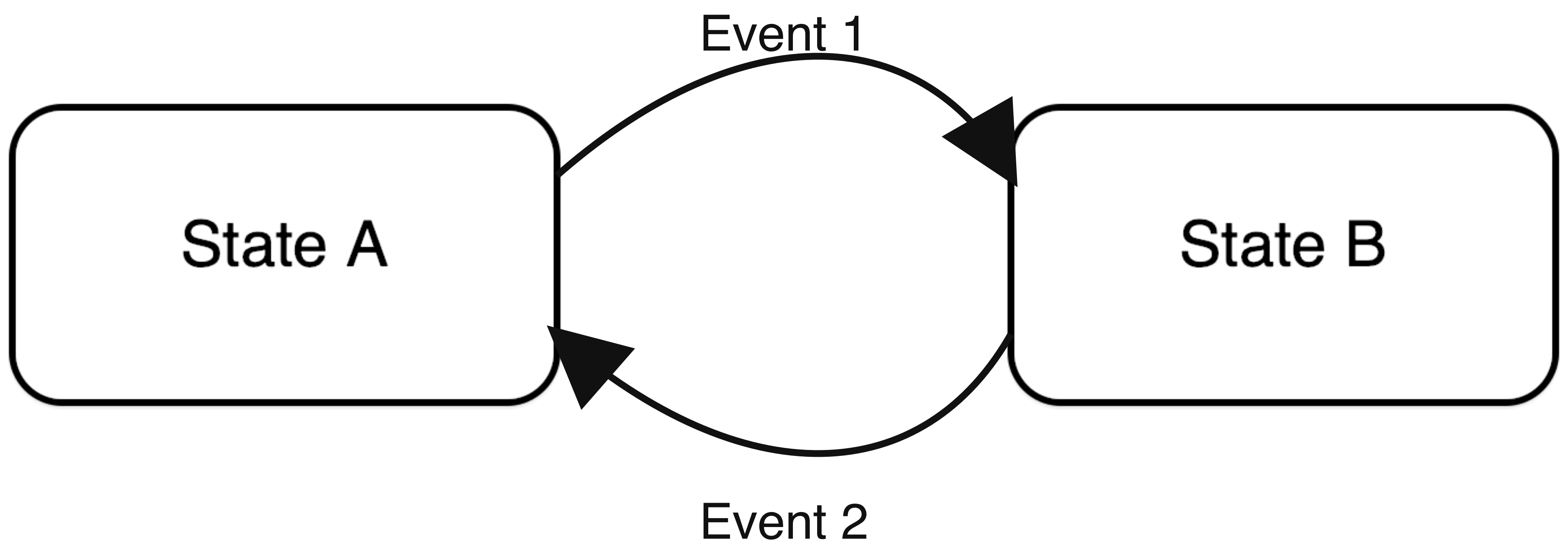
- Think of a Module as a **State Machine**
  - Events trigger *state transitions*
  - Incoming event / Trigger some other event

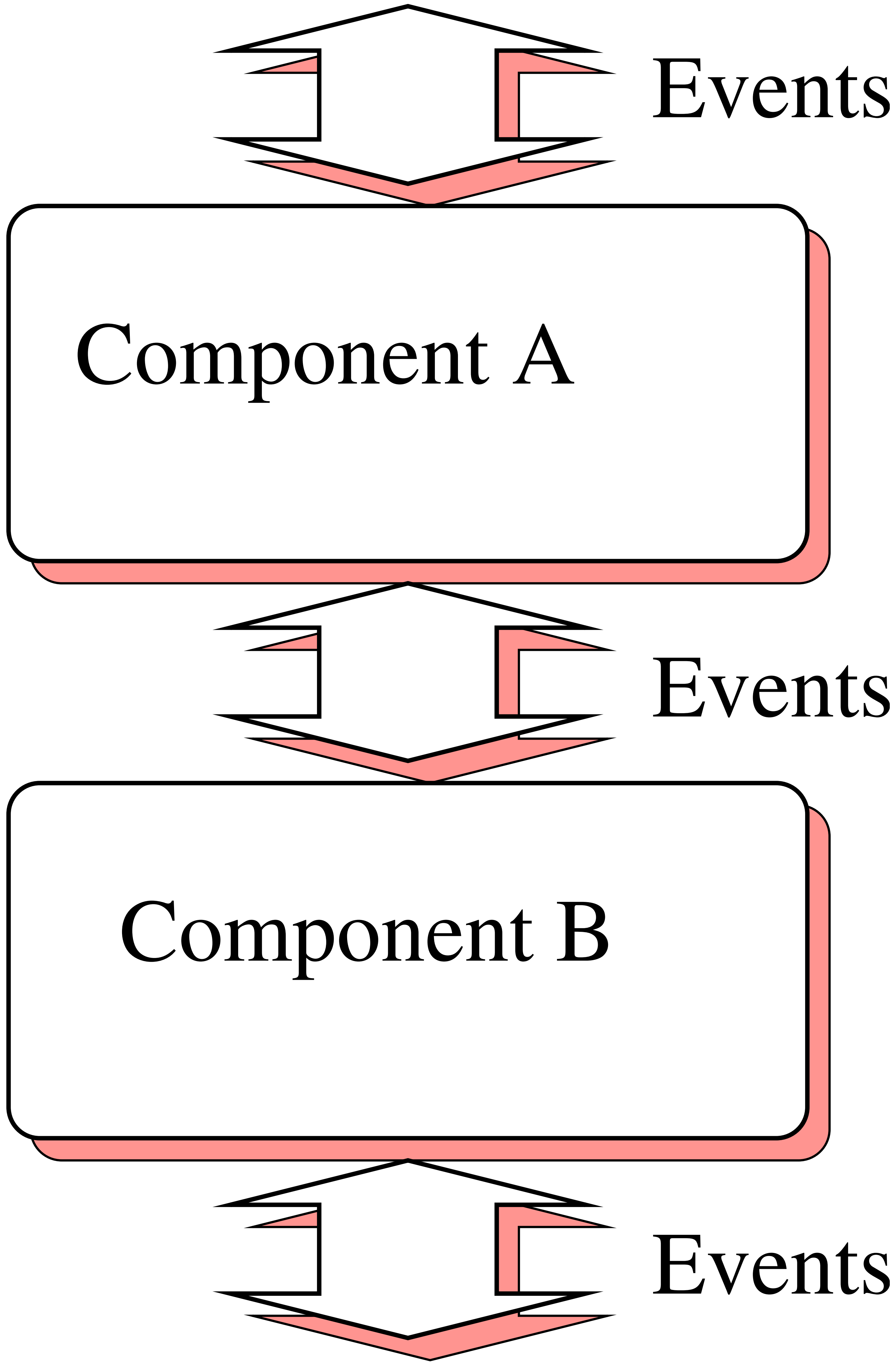
$$\textbf{upon event} \ \langle\, co_1, Event_1 \mid att_1^1, att_1^2, \ldots \,\rangle \ \textbf{do}$$
$$\quad \text{do something;}$$
$$\quad \textbf{trigger} \ \langle\, co_2, Event_2 \mid att_2^1, att_2^2, \ldots \,\rangle;$$



Event Handlers

- Think of a Module as a **State Machine**
  - Events trigger *state transitions*
  - Incoming event / Trigger some other event



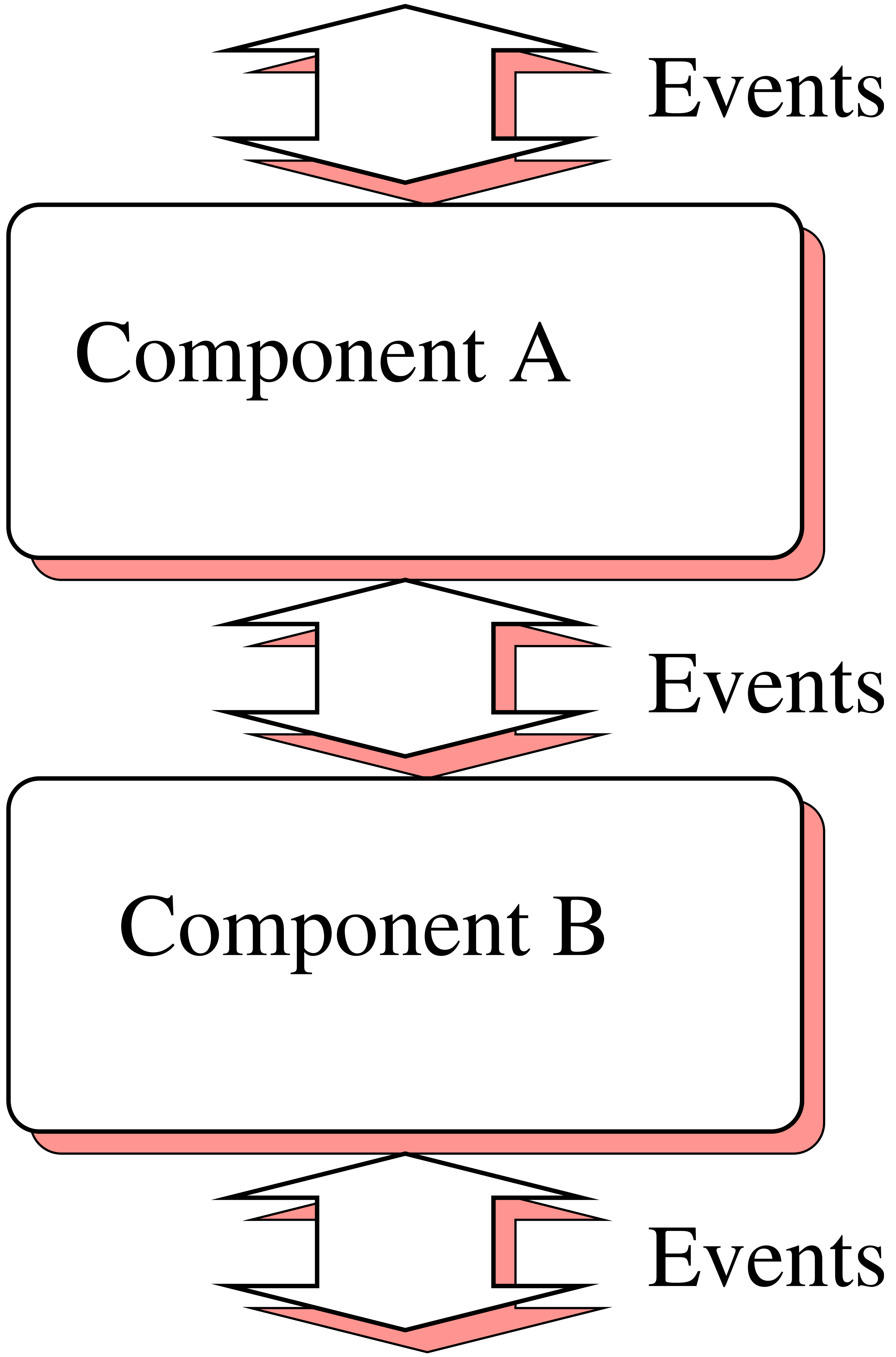


1.4 Software Components

Event Handlers

- Events may carry information in *attributes*
- Events are *processed in atomically*, i.e.,
  - no two events are processed concurrently

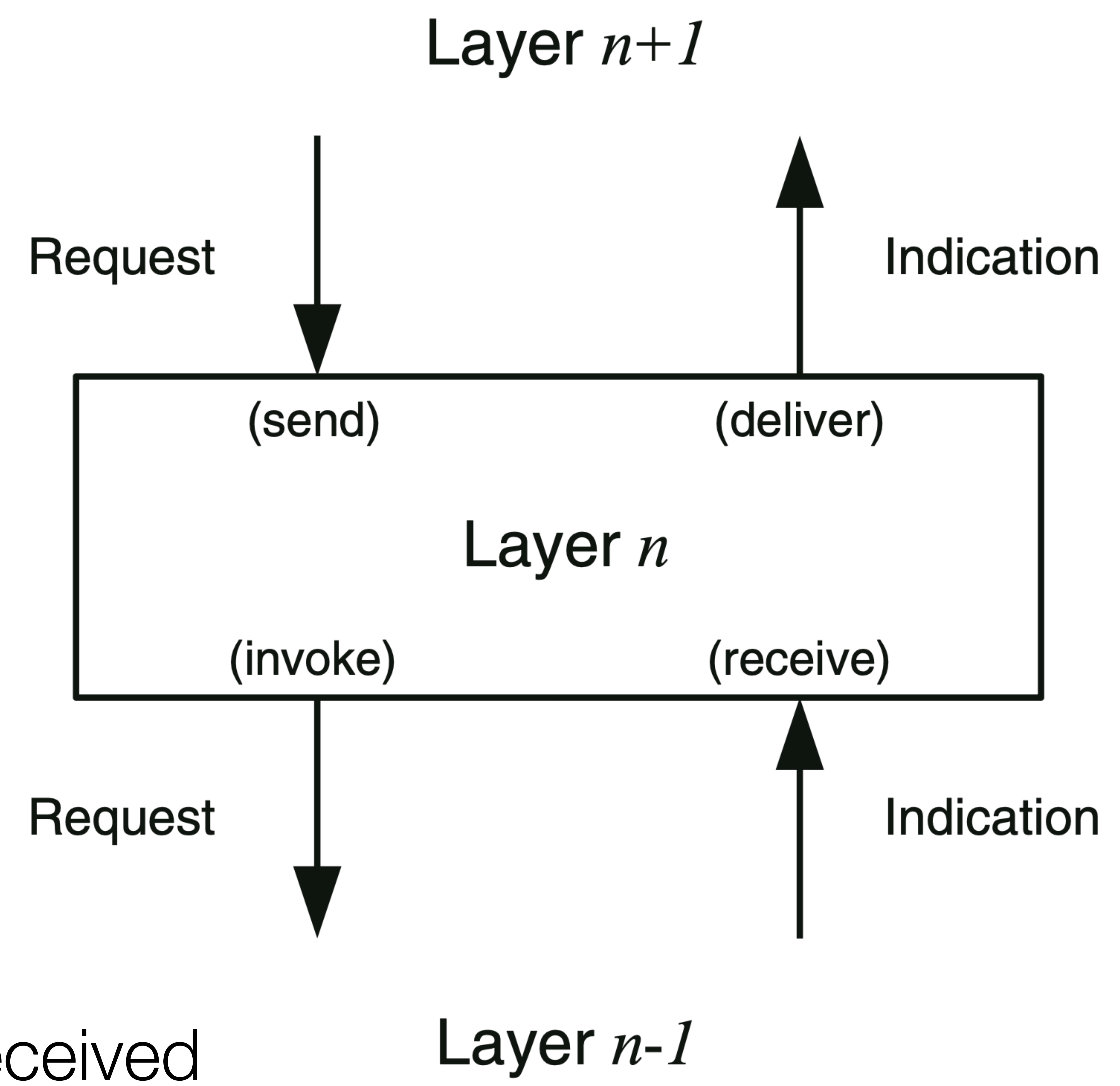
upon event $\langle\, co_1, Event_1 \mid att_1^1, att_1^2, \ldots \rangle$ **do**
do something;
trigger $\langle\, co_2, Event_2 \mid att_2^1, att_2^2, \ldots \rangle$;



1.4 Software Components

Programming Interface

- **Request** events (downcall)
  - ask for service from another module
  - e.g., to broadcast a message
- **Indication** events (upcall)
  - used to deliver information to another module
  - e.g., to deliver a message in-order that was received out-of order

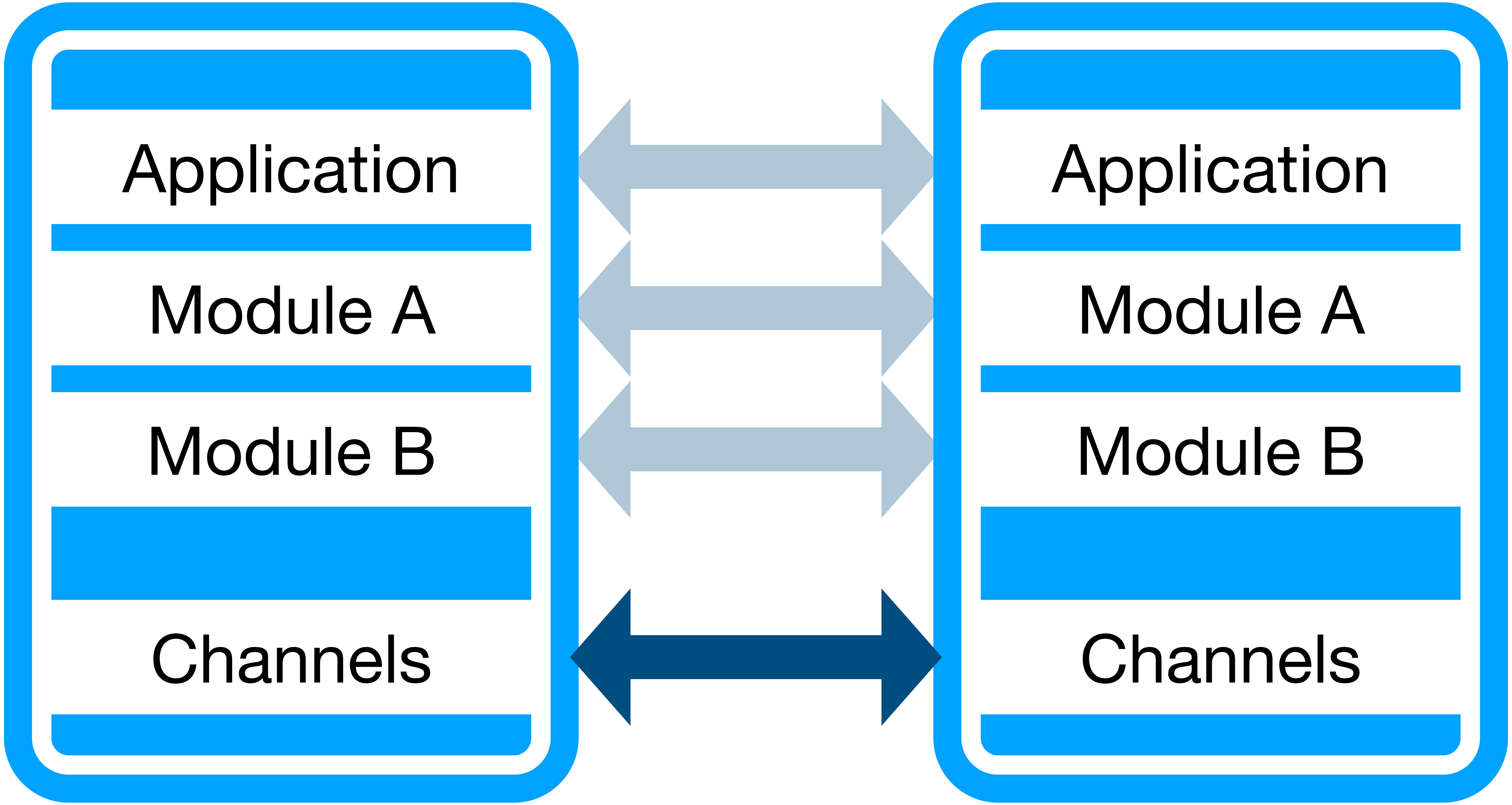


Interaction Between Processes

Module Interaction Between Processes

- Modules interact with corresponding modules on **peer processes**
- May have multiple instances (copies) of a module in one process

1.4 Software Components



Communication Between Modules

- Asynchronous interaction
  - **non-blocking**: send request *don’t wait* for reply/result
- Synchronous interaction
  - **blocking**: send request *wait* for reply/result

Textbook

Joke time!


Synchronous
SIGH...
TAP
TAP
We're all just waiting for Bob...”
Asynchronous
Bob will get back to us... maybe.”

Algorithmic Terminology

Protocol Module Examples

1.4 Software Components

Module 1.1: Interface and properties of a job handler

Module:

Name: JobHandler, **instance** *jh*.

Events:

Request: $\langle\, jh,\ Submit \mid job \,\rangle$: Requests a job to be processed.

Indication: $\langle\, jh,\ Confirm \mid job \,\rangle$: Confirms that the given job has been (or will be) processed.

Properties:

JH1: *Guaranteed response:* Every submitted job is eventually confirmed.

1.4 Software Components

Algorithm 1.1: Synchronous Job Handler

Implements:

JobHandler, **instance** *jh*.

upon event $\langle$ *jh*, *Submit* | *job* $\rangle$ **do**

process(*job*);

trigger $\langle$ *jh*, *Confirm* | *job* $\rangle$;

1.4 Software Components

Algorithm 1.2: Asynchronous Job Handler

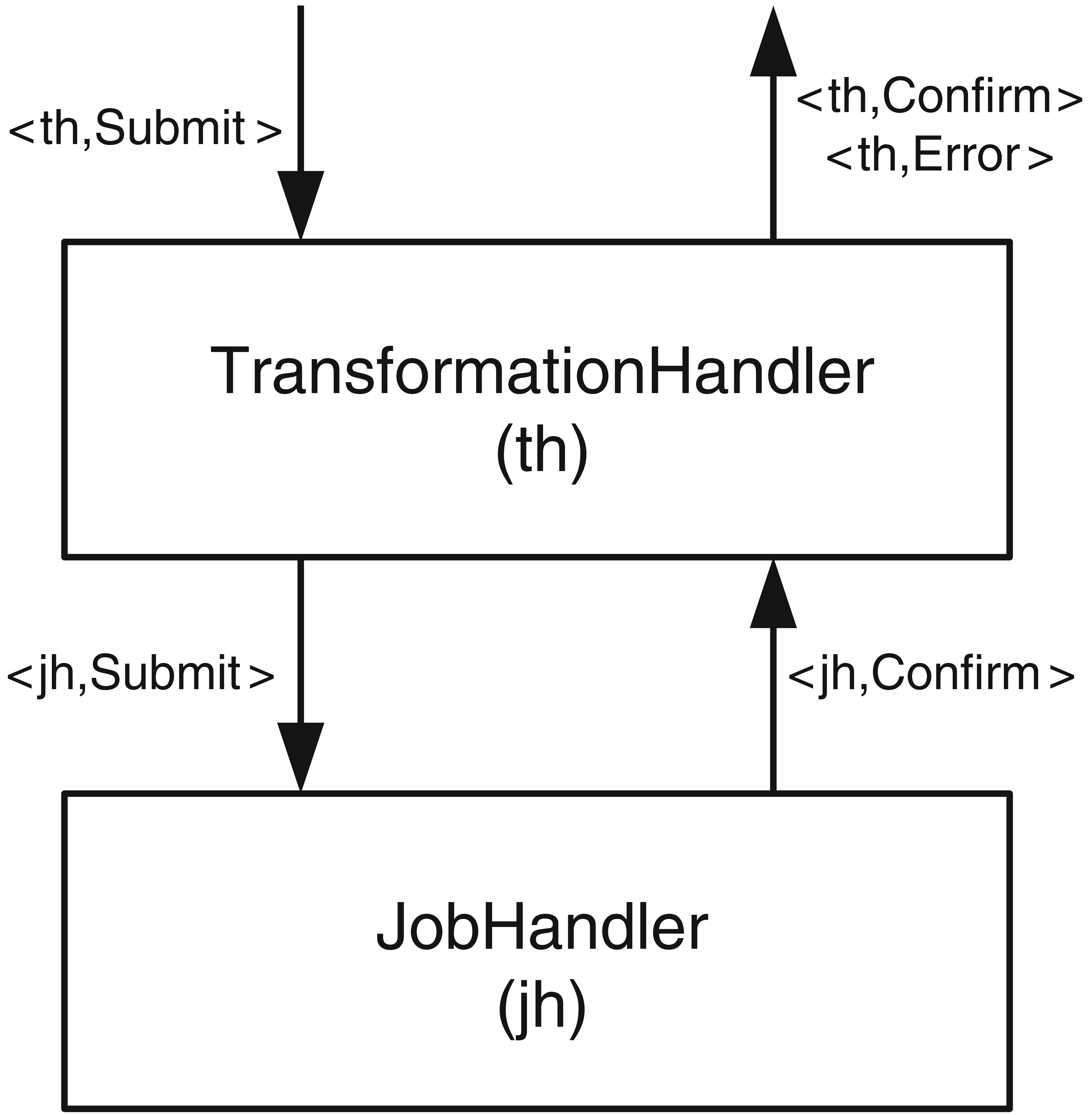
Implements:
    JobHandler, **instance** *jh*.

upon event $\langle\, jh, Init\,\rangle$ **do**
    $buffer := \emptyset;$

upon event $\langle\, jh, Submit \mid job\,\rangle$ **do**
    $buffer := buffer \cup \{job\};$
    trigger $\langle\, jh, Confirm \mid job\,\rangle;$

upon $buffer \neq \emptyset$ **do**
    $job := \text{selectjob}(buffer);$
    $\text{process}(job);$
    $buffer := buffer \setminus \{job\};$

1.4 Software Components



1.4 Software Components

Module 1.2: Interface and properties of a job transformation and processing abstraction

Module:

Name: TransformationHandler, **instance** *th*.

Events:

Request: $\langle$ *th*, *Submit* | *job* $\rangle$: Submits a job for transformation and for processing.

Indication:$\langle$ *th*, *Confirm* | *job* $\rangle$: Confirms that the given job has been (or will be) transformed and processed.

Indication:$\langle$ *th*, *Error* | *job* $\rangle$: Indicates that the transformation of the given job failed.

1.4 Software Components

Algorithm 1.3: Job-Transformation by Buffering

Implements:
TransformationHandler, **instance** *th*.

Uses:
JobHandler, **instance** *jh*.

upon event $\langle$ *th*, *Init* $\rangle$ **do**
  $top := 1$;
  $bottom := 1$;
  $handling := \text{FALSE}$;
  $buffer := [\perp]^M$;

upon event $\langle$ *th*, *Submit* | *job* $\rangle$ **do**
  if $bottom + M = top$ **then**
    trigger $\langle$ *th*, *Error* | *job* $\rangle$;
  else
    $buffer[top \bmod M + 1] := job$;
    $top := top + 1$;
    trigger $\langle$ *th*, *Confirm* | *job* $\rangle$;

upon $bottom < top \wedge handling = \text{FALSE}$ **do**
  $job := buffer[bottom \bmod M + 1]$;
  $bottom := bottom + 1$;
  $handling := \text{TRUE}$;
  trigger $\langle$ *jh*, *Submit* | *job* $\rangle$;

upon event $\langle$ *jh*, *Confirm* | *job* $\rangle$ **do**
  $handling := \text{FALSE}$;

Joke time!

“Distributed systems would be easy —

"Distributed systems would be easy — if time existed."

Protocol vs Distributed Algorithm

1.5 Classes of Algorithms

To Provide Service

- A module at a process
    - Executes one or more **rounds** of message exchanges with
    - Peer modules at other/remote processes

1.5 Classes of Algorithms

Protocol is …

- The behavior of each peer characterized by
  - the set of messages each peer is capable of producing and accepting,
  - the format of these messages, and
  - the legal sequences of messages

Protocol's Purpose is to …

- Ensure the execution of some **distributed algorithm.**
- Concurrent execution of different sequences of steps
  - that ensure the provision of the desired service

1.5 Classes of Algorithms

Classes of Algorithmic Solutions

- Book provides insight into how
  - **Failure assumptions**, the **environment**, the system parameters, and
  - other design choices *affect the algorithm design* …
- Several different classes of algorithmic solutions to the
  - distributed programming abstractions

1.5 Classes of Algorithms

Classes of Algorithmic Solutions

1. **Fail-stop**: processes can *fail by crashing* but the crashes can be *reliably detected*
2. **Fail-silent**: process crashes can *never be reliably detected*
3. **Fail-noisy**: crashes can be detected, but not always in an accurate manner
4. **Fail-recovery**: processes can crash and later recover and participate in the algorithm
5. **Fail-arbitrary**: processes can deviate arbitrarily from the protocol specification and act in malicious, adversarial ways
6. **Randomized** algorithms: processes may make probabilistic choices

Questions?