Lecture 23a:

Under the Hood, Part 1: Implementing Message Passing

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2018

Today's Theme

A. Y. OWEN

Teenage Boy Showing the Engine of His First Car, a 1951 Mercury

The LIFE Picture Collection

Message passing model (abstraction)

- Threads operate within their own private address spaces
- Threads communicate by sending/receiving messages
  - send: specifies recipient, buffer to be transmitted, and optional message identifier ("tag")
  - receive: sender, specifies buffer to store data, and optional message identifier
  - Sending messages is the only way to exchange data between threads 1 and 2

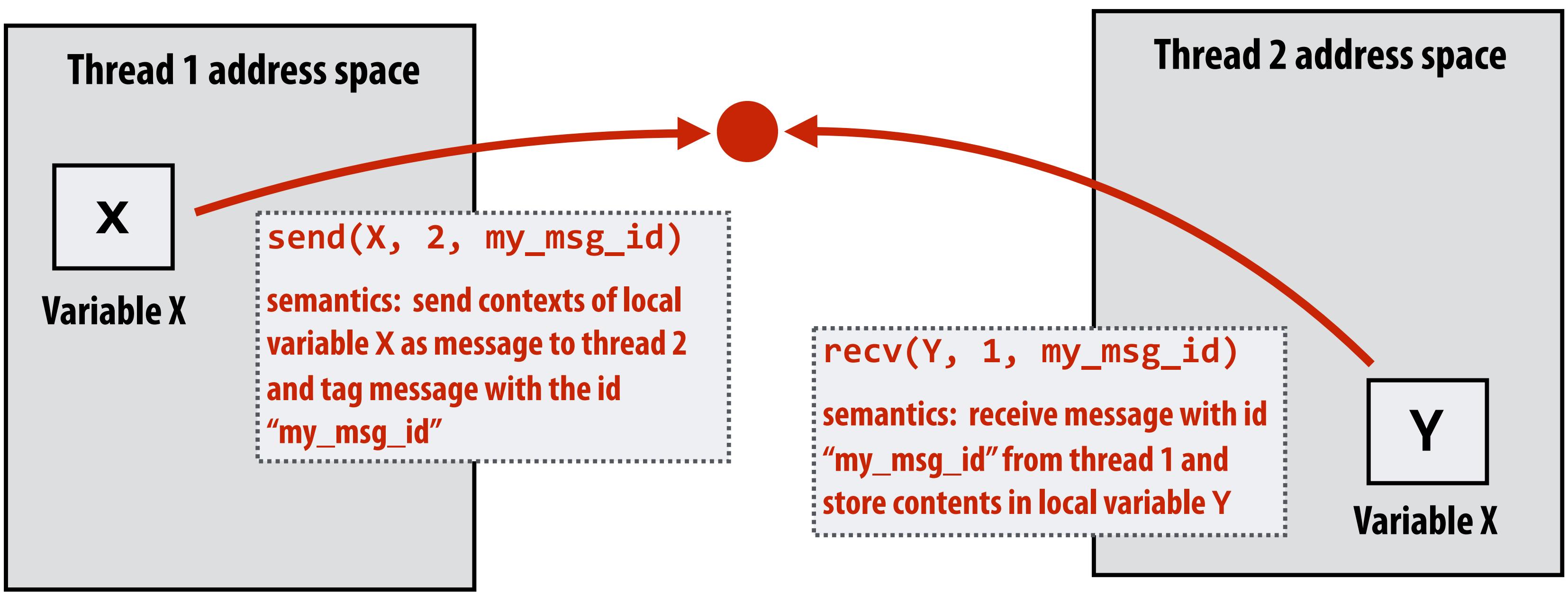
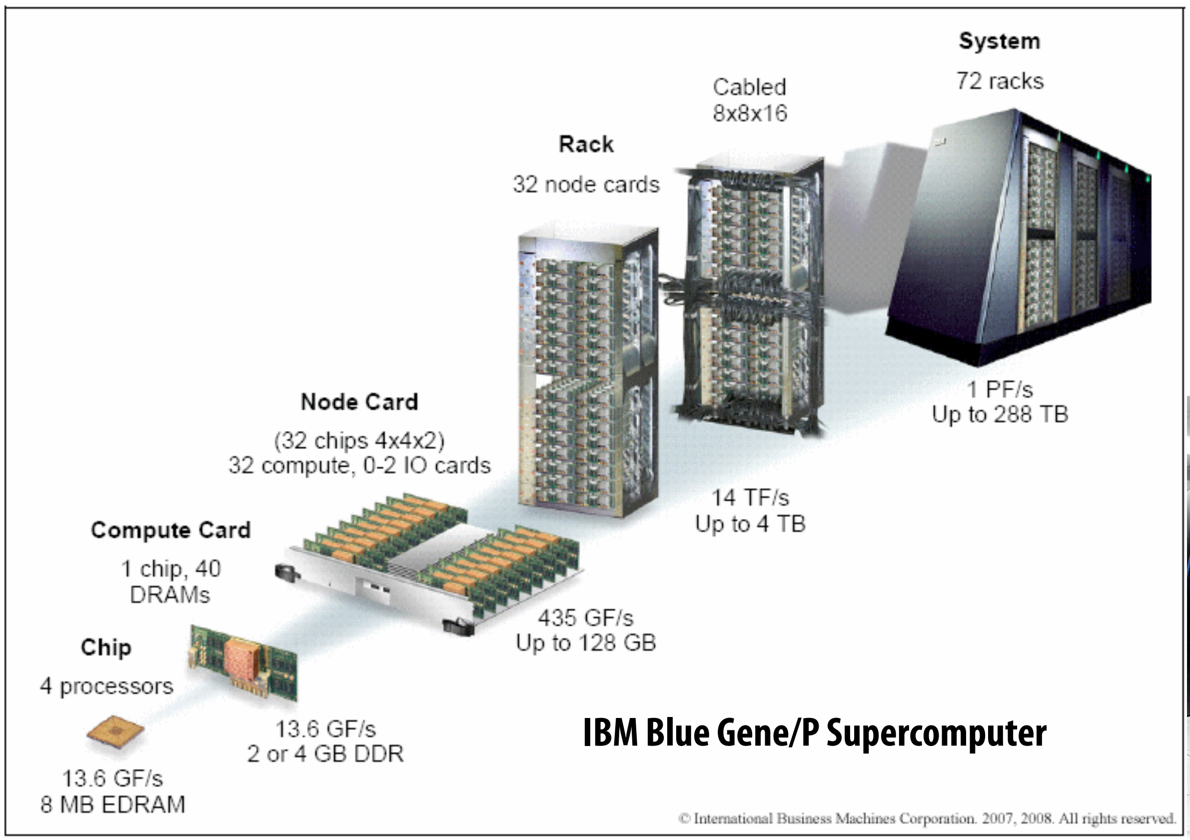


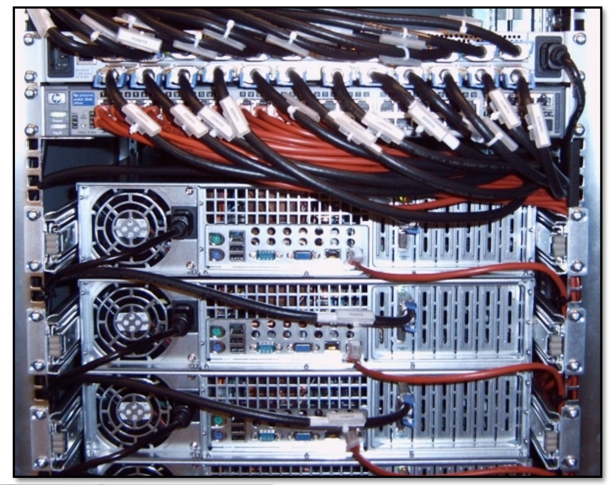
Illustration adopted from Culler, Singh, Gupta

Message passing systems

- Popular software library: **MPI** (message passing interface)
- Hardware need not implement system-wide loads and stores to execute message passing programs (need only be able to communicate messages)
  - Can connect commodity systems together to form large parallel machine (message passing is a programming model for clusters)



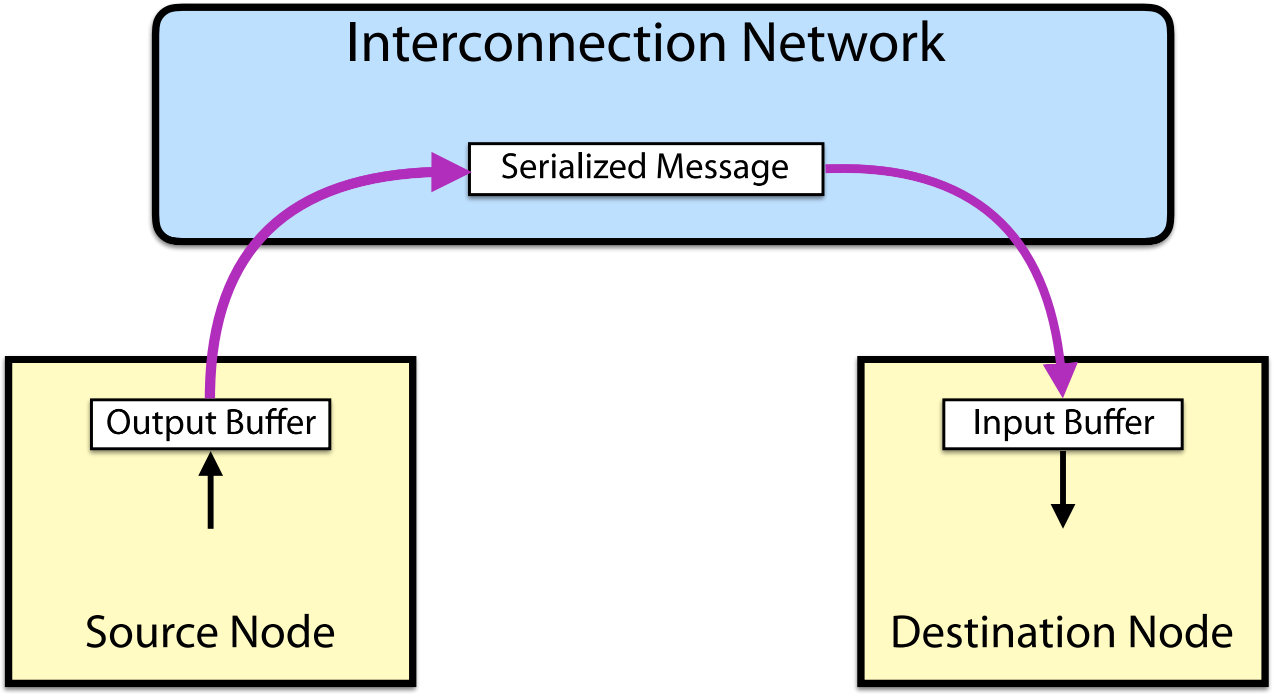
IBM Blue Gene/P Supercomputer



Cluster of workstations (Infiniband network)

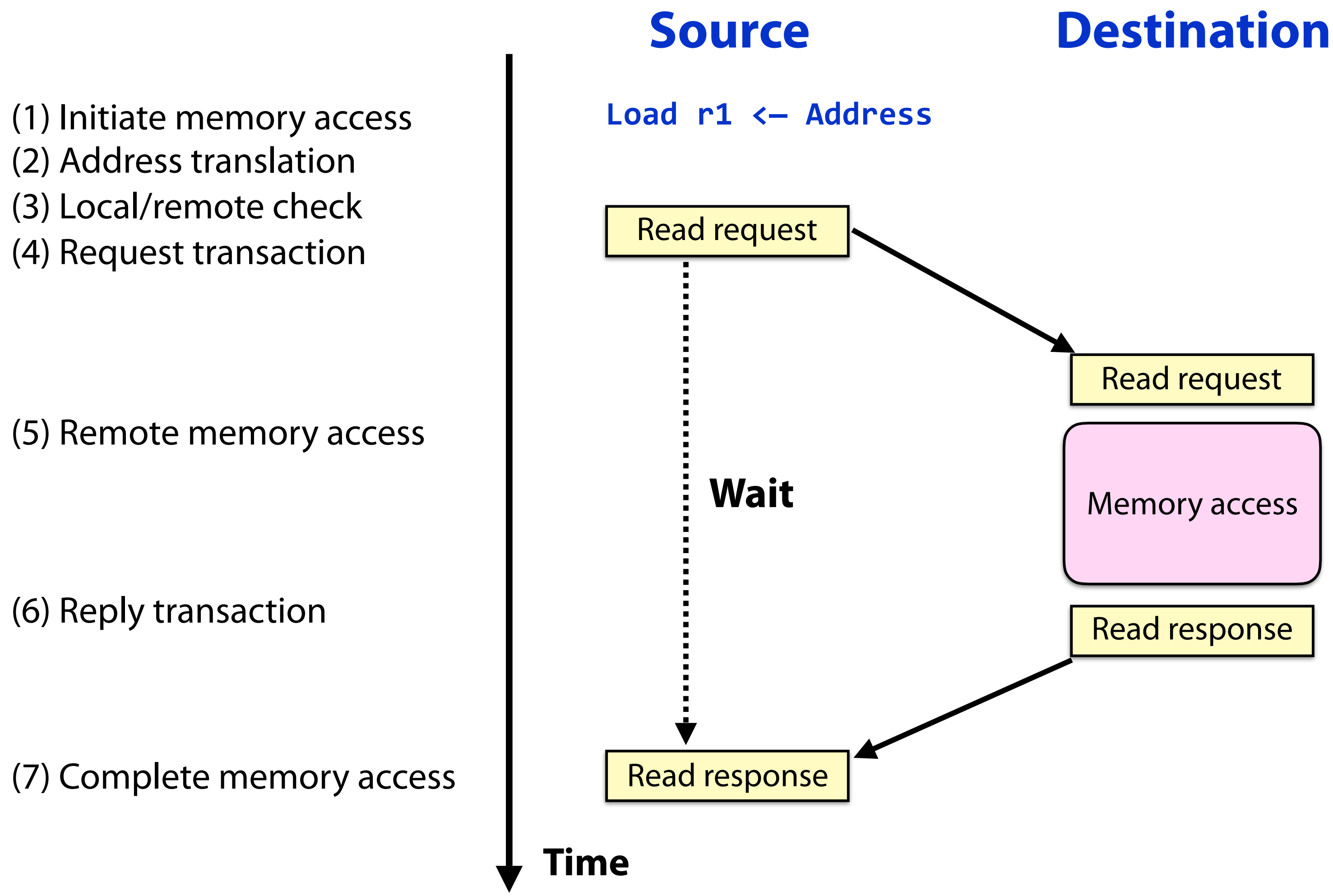
Image credit: IBM

Network Transaction



- **One-way transfer of information from a source output buffer to a destination input buffer**
  - **causes some action at the destination**
    - **e.g., deposit data, state change, reply**
  - **occurrence is not directly visible at source**

Shared Address Space Abstraction



- Fundamentally a two-way **request/response protocol**
  - writes have an acknowledgement

Key Properties of SAS Abstraction

- Source and destination addresses are specified by source of the request
  - a degree of logical coupling and trust
- No storage logically "outside the application address space(s)"
  - may employ temporary buffers for transport
- Operations are fundamentally request-response
- Remote operation can be performed on remote memory
  - logically does not require intervention of the remote processor

Message Passing Implementation Options

Synchronous:

- Send completes after matching receive and source data sent
- Receive completes after data transfer complete from matching send

Asynchronous:

- Send completes after send buffer may be reused

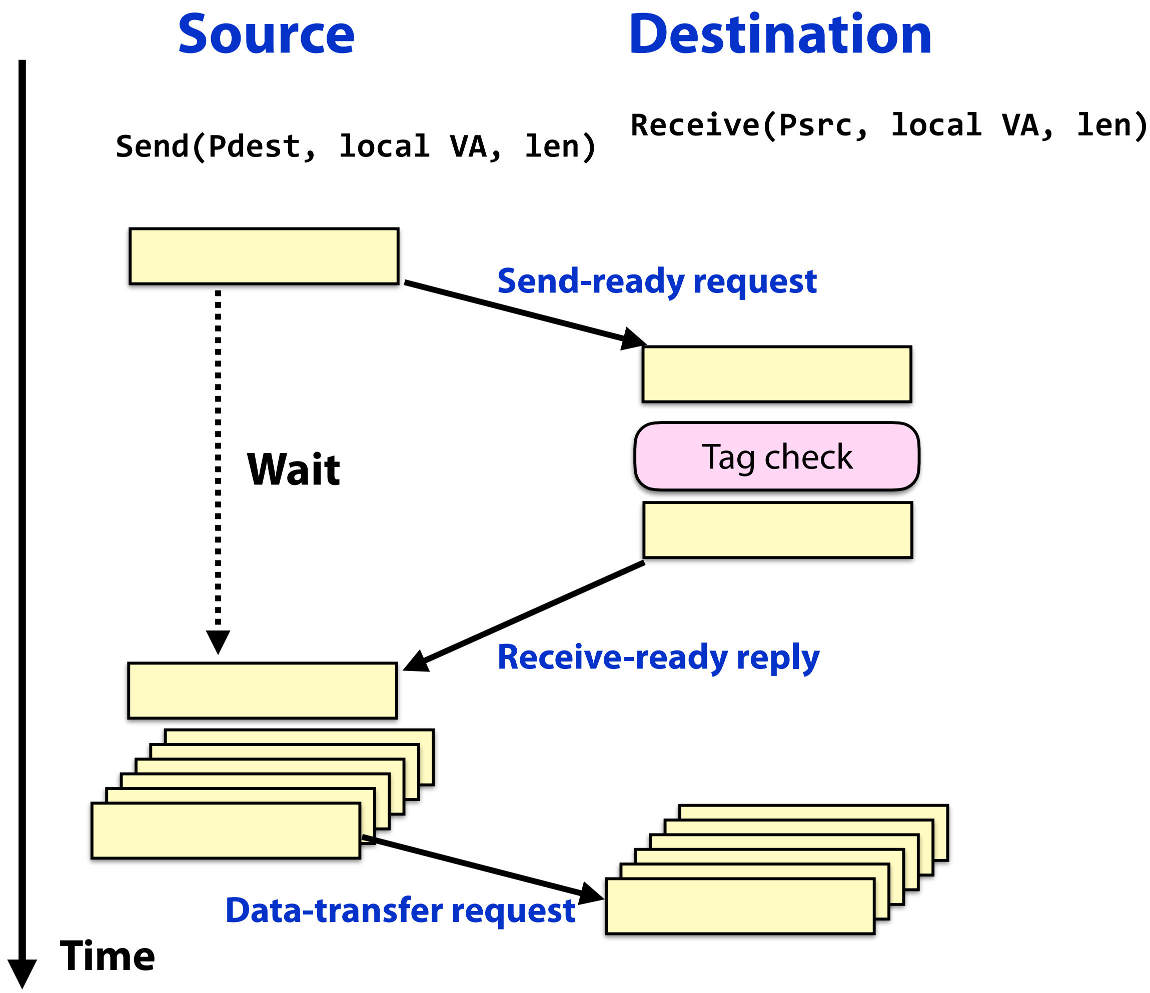
Synchronous Message Passing

(1) Initiate send
(2) Address translation
(3) Local/remote check
(4) Send-ready request

(5) Remote check for posted receive (assume success)

(6) Reply transaction

(7) Bulk data transfer
Source VA —> Dest VA

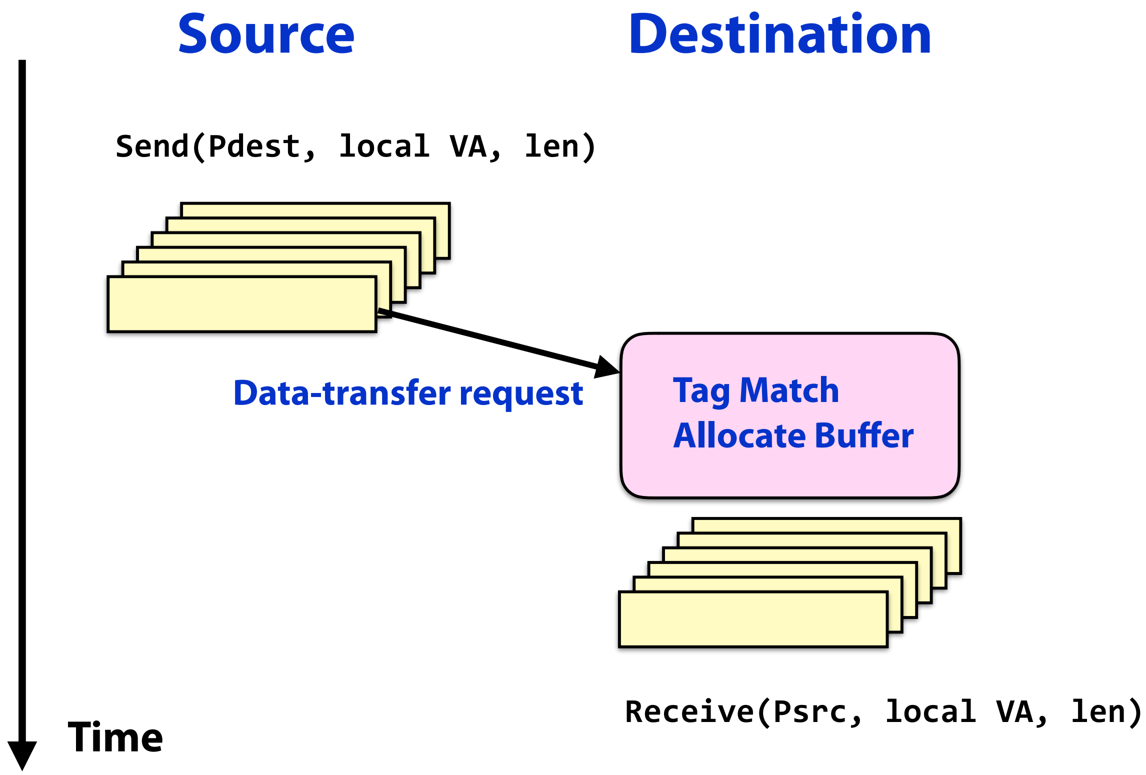


- **Data is not transferred until target address is known**
  - **Limits contention and buffering at the destination**
- **Performance?**

Asynchronous Message Passing: Optimistic

(1) Initiate send
(2) Address translation
(3) Local/remote check
(4) **Send data**

(5) Remote check for posted receive; on fail, **allocate data buffer**



- **Good news:**
  - source does not stall waiting for the destination to receive
- **Bad news:**
  - **storage is required within the message layer** (?)

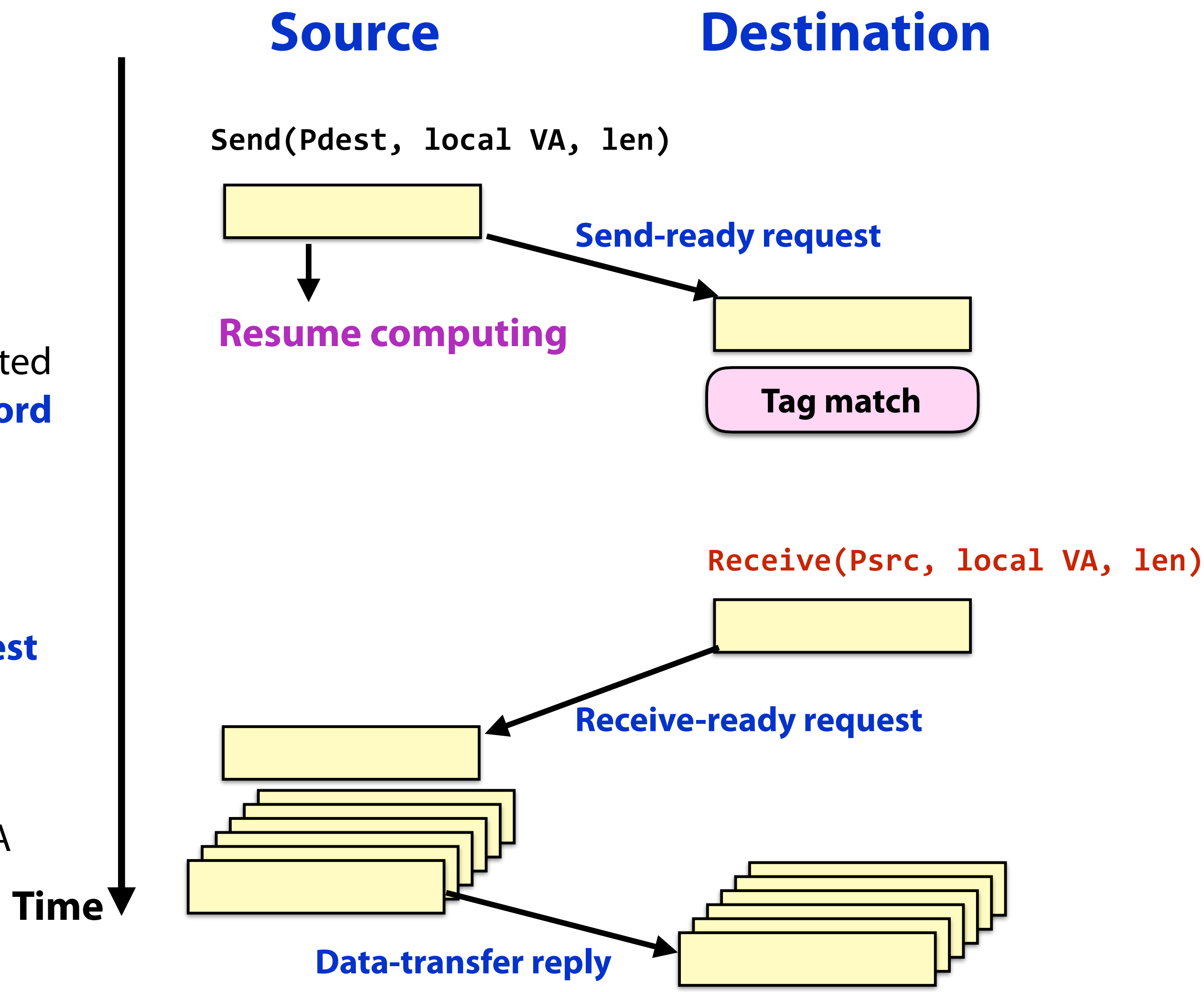
Asynchronous Message Passing: Conservative

(1) Initiate send
(2) Address translation
(3) Local/remote check
(4) Send-ready request

(5) Remote check for posted receive (assume fail); **record send-ready**

(6) Receive-ready request

(7) Bulk data reply
Source VA —> Dest VA



- **Where is the buffering?**
- **Contention control? Receiver-initiated protocol?**
- **What about short messages?**

Key Features of Message Passing Abstraction

- Source knows send address, destination knows receive address
  - after handshake they both know both
- Arbitrary storage "outside the local address spaces"
  - may post many sends before any receives
- Fundamentally a 3-phase transaction
  - includes a request / response
  - can use optimistic 1-phase in limited "safe" cases
    - credit scheme

Challenge: Avoiding Input Buffer Overflow

- This requires flow-control on the sources
- Approaches:
  1. Reserve space per source (credit)
     - when is it available for reuse? (utilize ack messages?)
  2. Refuse input when full
     - what does this do to the interconnect?
       - backpressure in a reliable network
       - tree saturation? deadlock?
       - what happens to traffic not bound for congested destination?
  3. Drop packets (?)
  4. ???

Challenge: Avoiding Fetch Deadlock

- Must continue accepting messages, even when cannot source msgs
  - what if incoming transaction is a request?
    - each may generate a response, which cannot be sent!
    - what happens when internal buffering is full?

Approaches:

1. Logically independent request/reply networks
   - physical networks
   - virtual channels with separate input/output queues
2. Bound requests and reserve input buffer space
   - K(P-1) requests + K responses per node
   - service discipline to avoid fetch deadlock?
3. NACK on input buffer full
   - NACK delivery?

Implementation Challenges: Big Picture

- **One-way transfer** of information
- **No global knowledge**, nor global control
  - barriers, scans, reduce, global-OR give fuzzy global state
- **Very large number of concurrent transactions**
- Management of **input buffer resources**
  - many sources can issue a request and over-commit destination before any see the effect
- **Latency is large enough that you are tempted to "take risks"**
  - e.g., optimistic protocols; large transfers; dynamic allocation