WoTUG20 Presentation

IMPLEMENTING HIGHER-ORDER CONCURRENCY IN JAVA

Erik Demaine
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
eddemaine@uwaterloo.ca

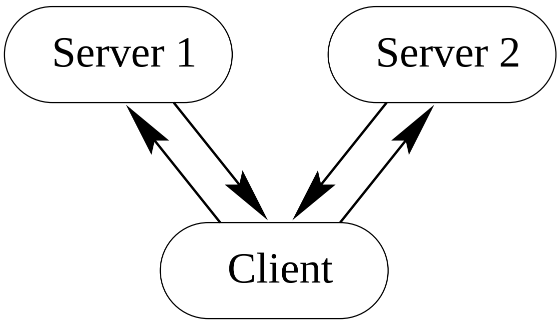
April 14, 1997

OUTLINE

- Higher-order concurrency
- Implementing in Java
  - Channels and morphing
  - Events
- Protocols
- Conclusion

HIGHER-ORDER CONCURRENCY

- Reppy proposed concurrent-programming model called **higher-order concurrency**
- Flexible construction of high-level communication constructs
- Example:



  - Define a call to the server (including protocol, etc.)
  - Don't want to recode for client to simultaneously talk to multiple servers
  - Especially if servers are of different types
  - First-class allows combination of protocols without recoding

HIGHER-ORDER CONCURRENCY: DETAILS

- Extension to Hoare's CSP
- Channels: unidirectional "pipes"
  - Can pass channels over channels
- Processes are dynamically created and garbage collected
- **Events**: synchronous communications on channels
  - Primitive events:
    - ☐ **transmit** (chan, data)
    - ☐ **receive** (chan)
  - Execute events via **sync** operation
  - Combinators:
    - ☐ **choose** (e_1, ..., e_n): selection between events
    - ☐ **wrap**, **wrapAbort**, **guard**: sequence of events
- Features are flexible; few needed

HIGHER-ORDER CONCURRENCY: ADVANTAGES

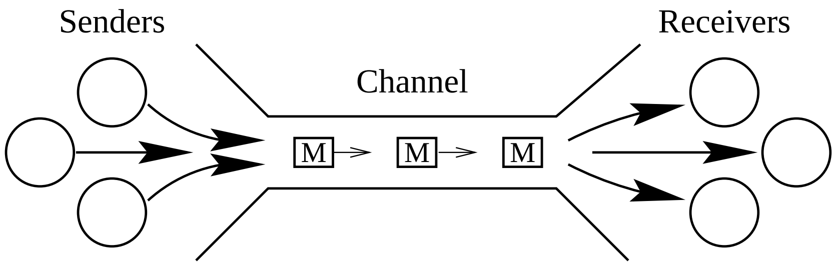
- Abstraction into **first-class events**
  - Define complex operations by functions returning events
  - Can treat new operations as if they were built-in
- Can implement
  - Buffered channels (asynchronous communication)
  - Multicast channels and thus multicast operations
  - Condition variables (write-once variables)
  - Ada-style rendezvous, plus allowing nested transactions
  - Locks and semaphores
  - Multilisp futures
- Even further: can selectively choose between Ada entry calls or obtaining locks, etc.
- Concurrent-programming language doesn't have to "guess" what concurrency structures might be needed

HIGHER-ORDER CONCURRENCY IN JAVA

- Java's suitability:
  - Designed for concurrent programming
    - ☐ Full subtyping
  - Garbage collection of objects
- Recently support for distributed computing (JDK 1.1.1)
- Threads can only communicate through mailboxes

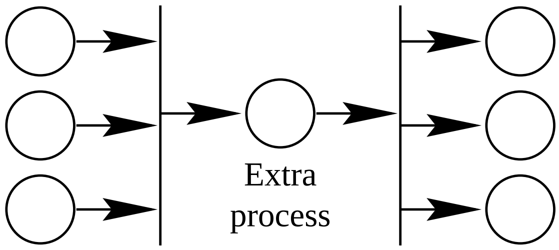
CHANNELS

- In higher-order concurrency, channels are many-to-many



- Difficult to maintain group membership in distributed system

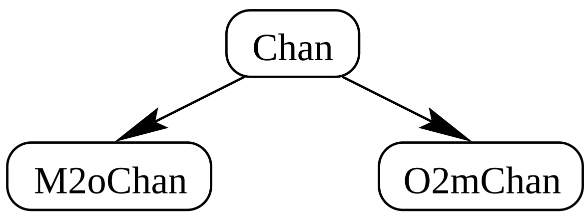
- Instead: many-to-one and one-to-many channels
  - **Owner**: the single receiver/sender
  - **Members**: other processes connected to channel
- If necessary, can build up many-to-many channels



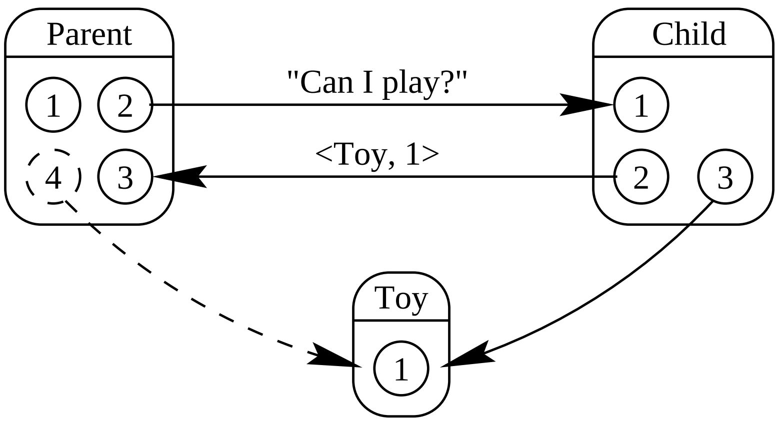
  - With higher-order concurrency, can treat these channels as built-in

CHANNELS

- Class hierarchy:



- Define ways to create channels and connect processes to them:
  - Any process p can dynamically create channel with owner p, either many-to-one or one-to-many
    - ☐ Call `new M2oChan ()`, or `new O2mChan ()`
  - When spawning, obtain two channels between self and child
    - ☐ `spawn` routine returns `Chan [ ]` array
    - ☐ Function defining thread code takes `Chan [ ]` argument
  - Other channel creation is done via message passing
    - ☐ `new Transmit (ch1, ch2).sync ();`
- Example:



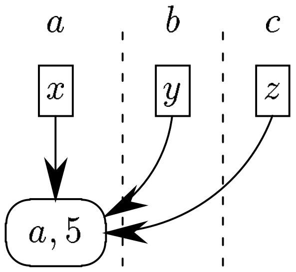
MORPHING CHANNELS: WHY MORPH?

- How to support sending channels over channels?

Three requirements:

1. Detect when there are no members
2. Detect when owner discards its references
3. The owner must remember the channel, even after (2)

- Cannot simply pass **Chan** reference:



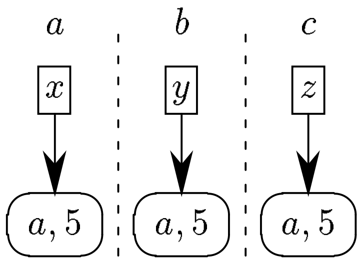
  - Inefficient in distributed system
  - Conditions (1) and (2) not satisfied

MORPHING CHANNELS: PROBLEMS

Three requirements:

1. Detect when there are no members
2. Detect when owner discards its references
3. The owner must remember the channel, even after (2)

- Hence must "morph" each time channel is communicated:



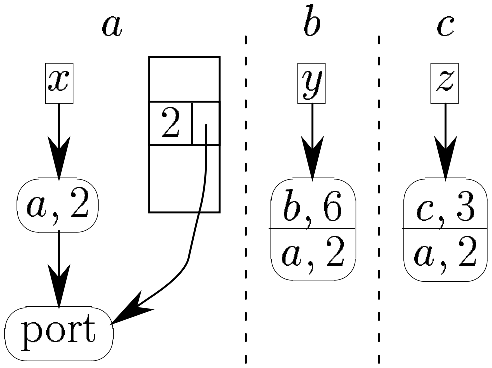
  - Can detect discards via **finalize ()** method
  - Cannot satisfy (3)

MORPHING CHANNELS: PORT SOLUTION

Three requirements:

1. Detect when there are no members
2. Detect when owner discards its references
3. The owner must remember the channel, even after (2)

- Solution: Split owner's channel into a **Chan** and a **Port**:



  - **Port** detects (1) and (2), persisting until both happen
  - Can find **Port** by looking up id
  - **Chan**s explicitly or implicitly have reference to **Port**

- Satisfies (1–3)

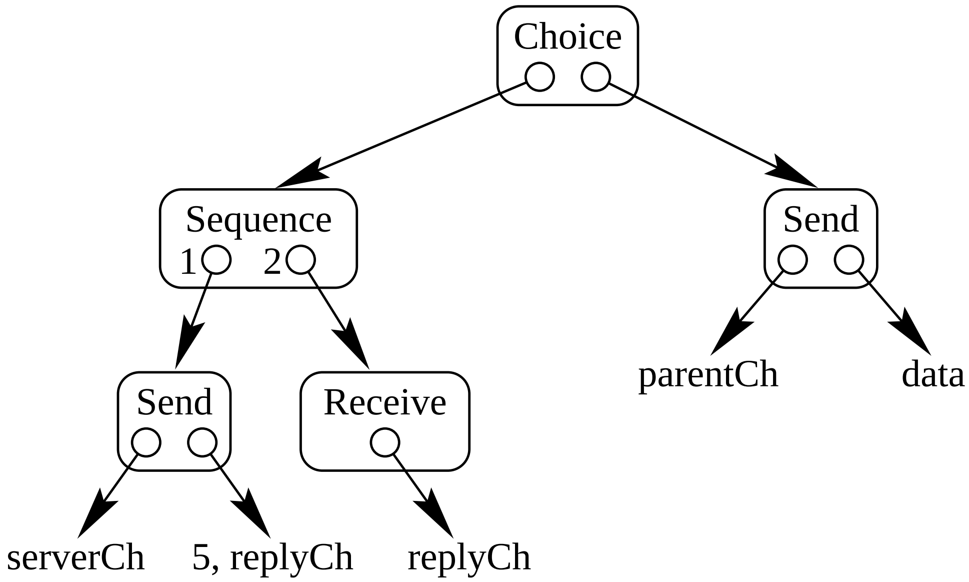
HOW TO MORPH CHANNELS

- User always communicates through channels
- Can check if object is a Chan via instanceof
- "Container" types
  - Must implement Morphing interface
  - Provide ways to traverse elements and recursively morph
- New: Object serialization
  - Allows traversing data structures portably

EVENTS

- Event $\longleftrightarrow$ tree
  - Internal nodes correspond to combinators
  - Leaves correspond to primitive events
  - External nodes correspond to channels or data items

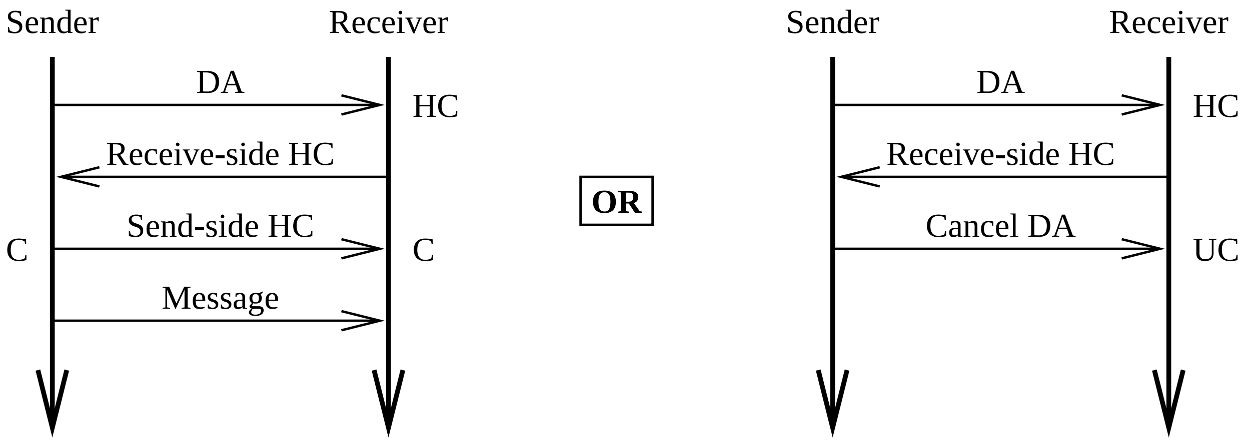
- Example:



PROTOCOLS

- How to execute an event?
  - Non-determinism: processes change their mind
- How to negotiate with potential partners to pick an operation such that
  - the partner chooses the complement of the operation
  - no deadlock occurs
  - fast
- Main difficulty: both sends and receives in **alt**
  - Not a problem in occam
- Problem has been examined before by Bornat (1984) and Knabe (1992)

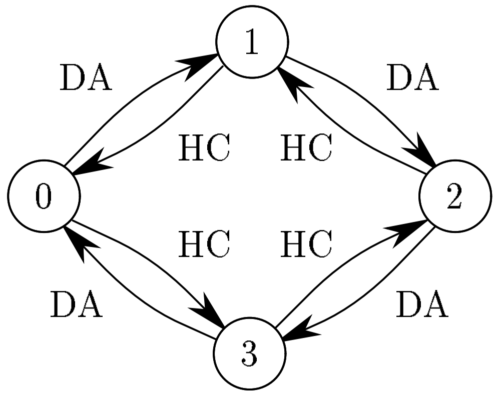
SIMPLE PROTOCOL



- Data available (DA) and want data (WD) messages
- Half-commit (HC) on receipt of DA or WD, if not half-committed
- Two HCs indicate a committed transaction (C)
- Cancel causes uncommit (UC); then try something else
- To unicast DA and WD messages to owner:
  - Use DA for sending on many-to-one channel
  - Use WD for receiving on one-to-many channel
- Performance
  - 4 message cycles
  - 3 cycles when many-to-one; message piggybacks second HC

DEADLOCK-FREE SIMPLE PROTOCOL

- Get potential deadlock if a cycle of DA (WD) messages



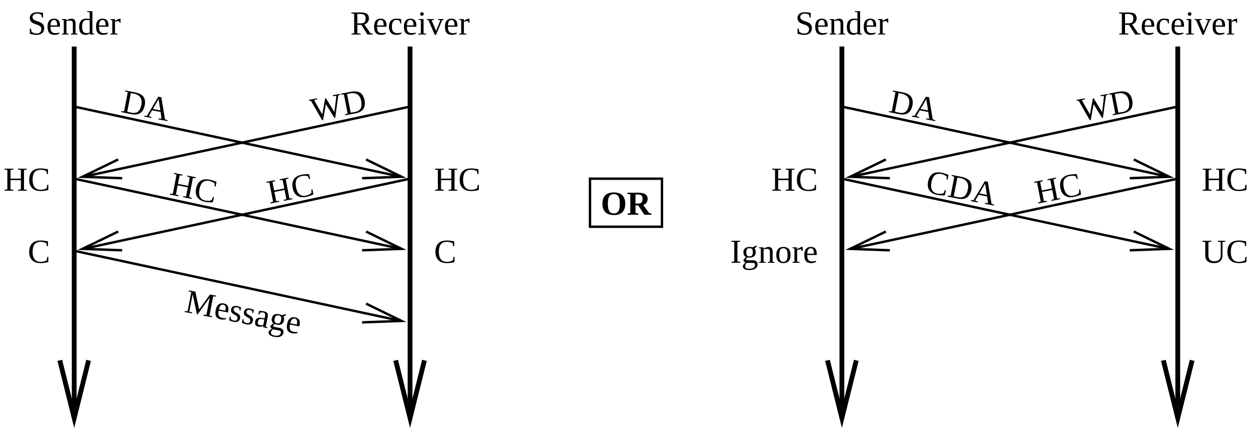

```
par i = 0 for 4
  alt
    (i+1)\4 ! ...
    (i+3)\4 ? ...
```

- Can use WD for many-to-one and DA for one-to-many
  - Multicast to all receivers (senders)
  - Group membership can be maintained at owner
  - Potential flexibility
- Use unique id of processes (totally ordered)
- Send WD/DA messages to processes with lower id only

  $\Rightarrow$ Cycles are broken
- Prevents deadlock

FAST PROTOCOL (with deadlock)

- 3 or 4 messages is too slow
- Goal: 2-message synchronous transfer (message + ACK)
- Solution: Use both DA and WD messages for every channel



- Performance
  - 3 message *cycles*
  - Just 2 cycles if message piggybacks send-side HC

DEADLOCK-FREE FAST PROTOCOL

- Must use deadlock avoidance
  - Can't selectively use DA and WD, since we need both
- Wait for deadlock to occur (with high probability)
- On timeout, if half-committed,
  - Guess there is a deadlock
  - Send everyone half-committed to me a "try something else" message
  - That is, force them to uncommit
- Problem: Everyone might force everyone else, possibly leading to the opposite cycle (e.g., DA instead of WD)
- Solution: Use same trick as simple protocol
  - Only force processes that have a lower id
  - This ensures that some process in every cycle won't be forced

    $\Rightarrow$ No deadlock

CONCLUSION

- Main features of higher-order concurrency implemented in Java:
  - First-class events
  - Garbage collection of processes
- Bringing ideas from one domain (programming languages) to another (message passing/parallel computing)
- Powerful concurrent programming can be achieved in Java
  - Java is currently slow
  - Overhead of abstraction
- Future
  - Distributed systems
  - Implement other protocols
  - Adaptive parameter modification and protocol switch