Lecture 18

Infopipes: an Abstraction for Information Flow

CSE 515 — Winter 2004

Our Application Domain

- Information-driven Applications
  - Transfer and process streams of information
- Examples:
  - distributed multimedia
    - streaming video and/or audio in real-time
  - environmental observation
    - Columbia River data: Forecast/Nowcast
  - weather forecasting

Application Requirements

- Applications need to direct *streams* of information
  - to the right place
  - at the right time
  - containing the right information
  - with the right Quality
    - Quality of Service is a compromise between application specific *desires* and available *resources*

Solution

- CORBA, DCE RPC, Java RMI, JGroups…

Solution

- ~~CORBA, DCE RPC, Java RMI, JGroups…~~

No!

Solution

- ~~CORBA, DCE RPC, Java RMI, JGroups…~~

No!

- These abstractions hide communication
- We want to reify communication

to reify = "to make the abstract real"

  - create concrete objects that represent communications abstraction
  - messages to these objects let us examine and change the properties of the communication link

Infopipes Reify Information Flows

- Infopipes reify communication

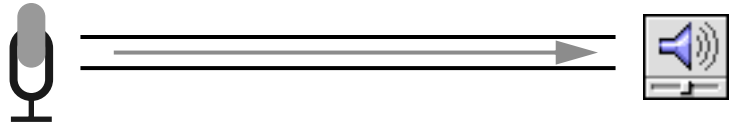
  …but at the *application* level

  not at the implementation level

- Example
  - bandwidth of Infopipe carrying compressed video
  - measured in frames per second, *not* bits per second
- Why?
  - If the application is going to do anything with flow information, it must be in application-level terms

Today…

- Suppose you want to build a digital phone or a digital camera…

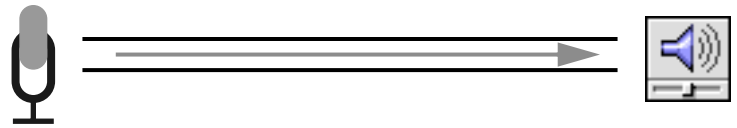


  – pretty easy!

    - just connect together some standard components

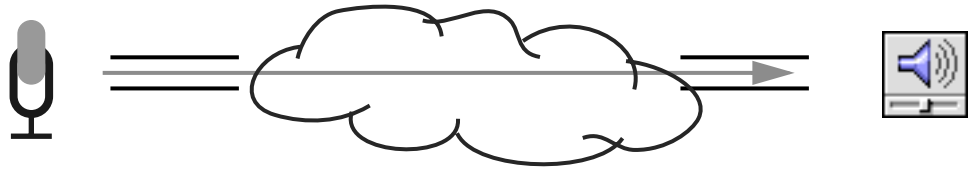
Today…

- Suppose you want to build a digital phone or a digital camera…



  – pretty easy!

and then make it work over the Internet



  – not so easy!

In the Infopipe World…

- Suppose you want to build a digital phone or a digital camera…

  – pretty easy!

    - just connect together some standard components

In the Infopipe World…

- Suppose you want to build a digital phone or a digital camera…

  – pretty easy!

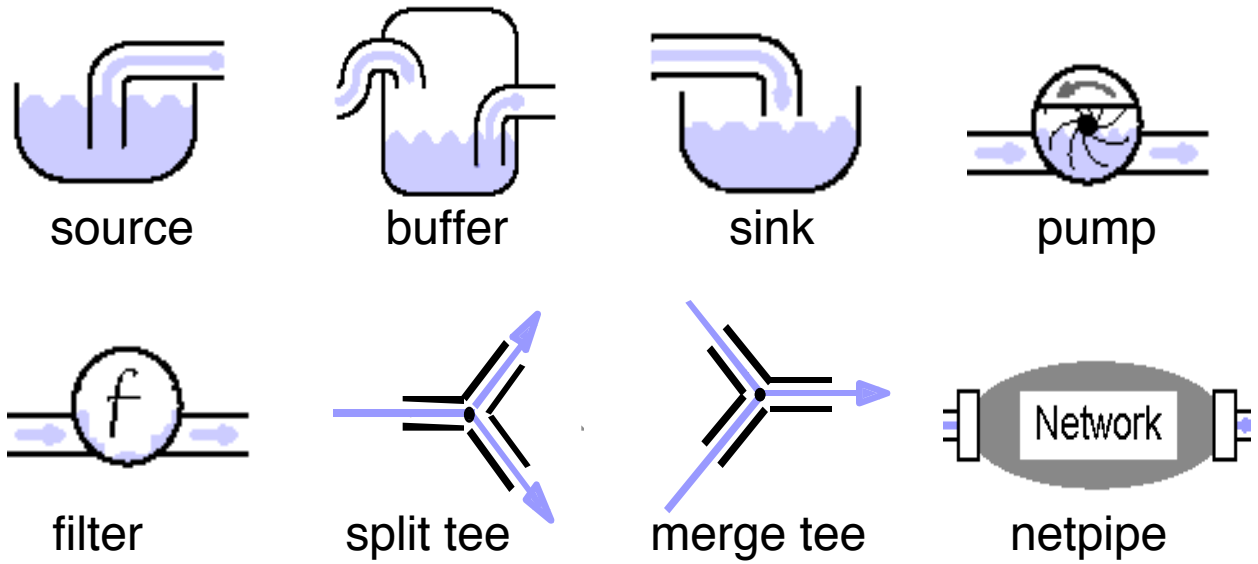
and then make it work over the Internet

  – just as easy!

What *are* Infopipes?

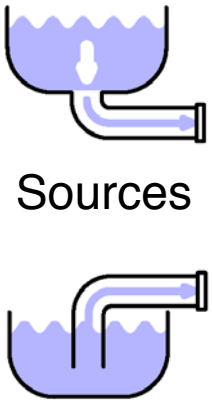
- System and distributed system abstraction
- Have well-defined characteristics, specifically, *rate*, *latency* and *jitter.*
- Compositional: the characteristics of a composite Infopipe can be calculated from those of its components
  - Seamless interconnection

Think “Plumbing”



- An Infopipe has zero or more “Inports” and zero or more “Outports”

Sources



- Devices that create data
  - cameras, microphones
  - environmental sensors
  - data mining queries on a database
  - POS transaction terminal
  - File stream
  - counter
  - random number generator
- May be periodic, sporadic, active or passive

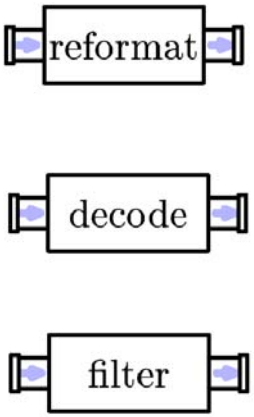
Pipes

- Transmission of Data
  - Network connections
  - Communication between address spaces (IPC)
  - Serial connections
  - Busses (USB, SCSI, backplanes …)

Pipe

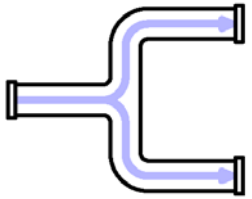
Transformers, Filters

- Perform computation on information stream
  - Mapper (*collect:*)
    - compression/decompression
    - labelling/delabelling
    - encryption/decryption
  - Dropper (*select:*)
    - resolution dropping
    - load sensitive dropping
  - parameterized by data and by external inputs

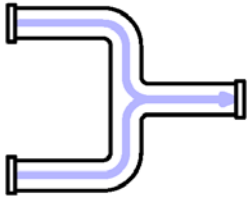


Tees

- Multicast tees (expansionist)
  - each input item is sent to all outports
- Switching tees (conservative)
  - an outport for each input item is selected, based on the item, history, external inputs



Split Tee

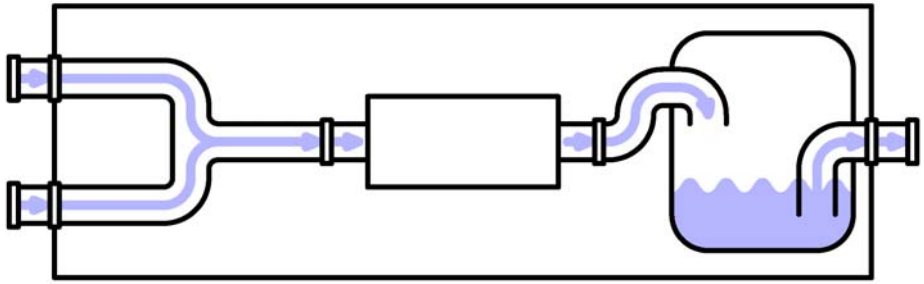


Join Tee

- Merge tees (retractive)
  - input items are zipped together
- RFP tees (conservative)
  - take packets as they come

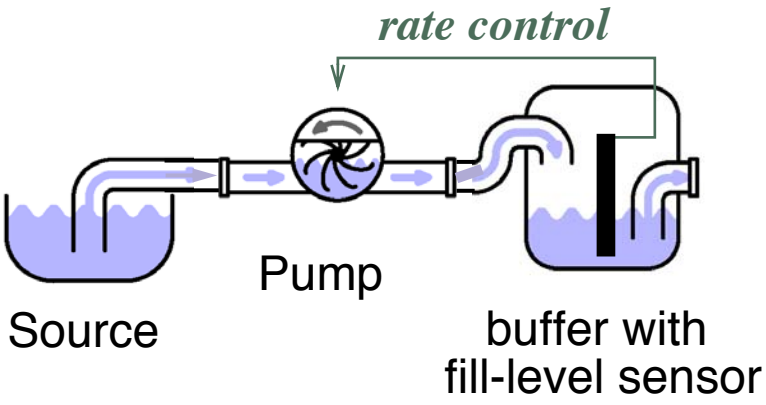
Composite Components

- Complex components can be built by putting a “black box” abstraction boundary around a Pipeline.



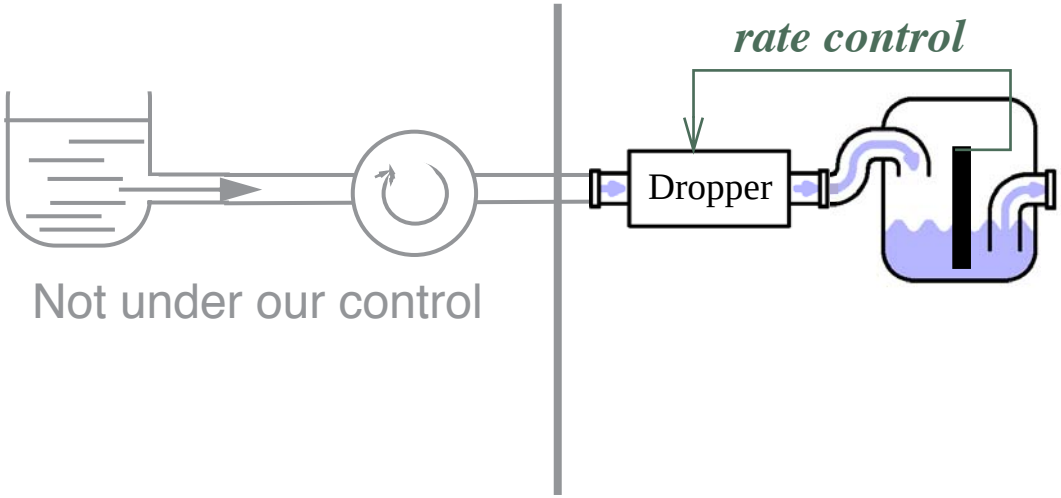
  – enables modularity and reuse

Feedback



- Rate of the pump is adjusted to keep buffer fill level within bounds.

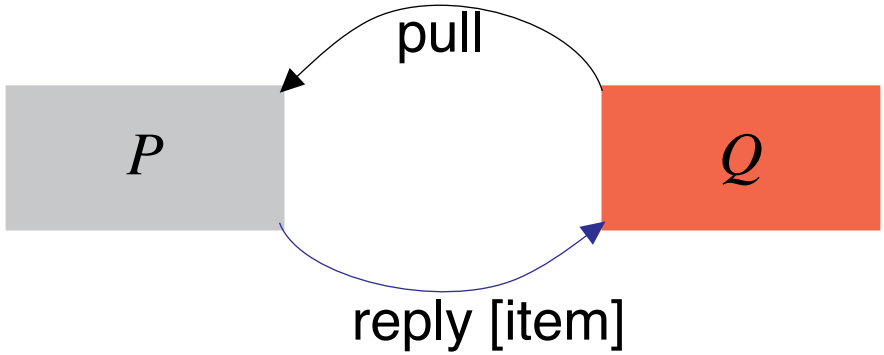
Feedback



- Feedback control drops packets selectively
  - avoid random dropping!
  - *e.g.*, video trans-coder labels packets with high-resolution imagery as “low priority”.

The Data Interface

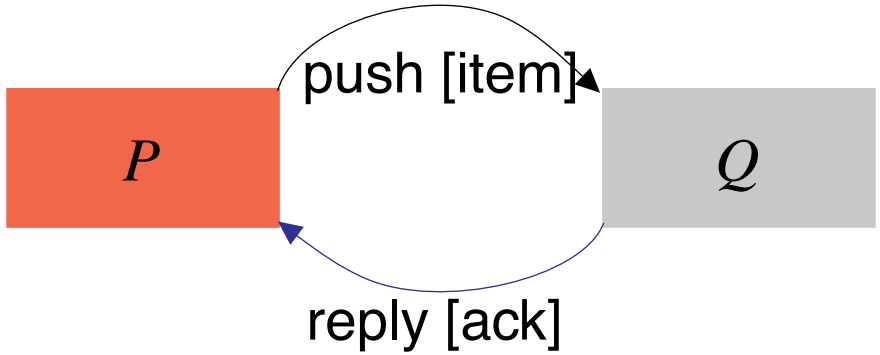
- Infopipes can understand *pull* messages



  - component *P* waits for its downstream neighbour *Q* to *pull* an item from *P*

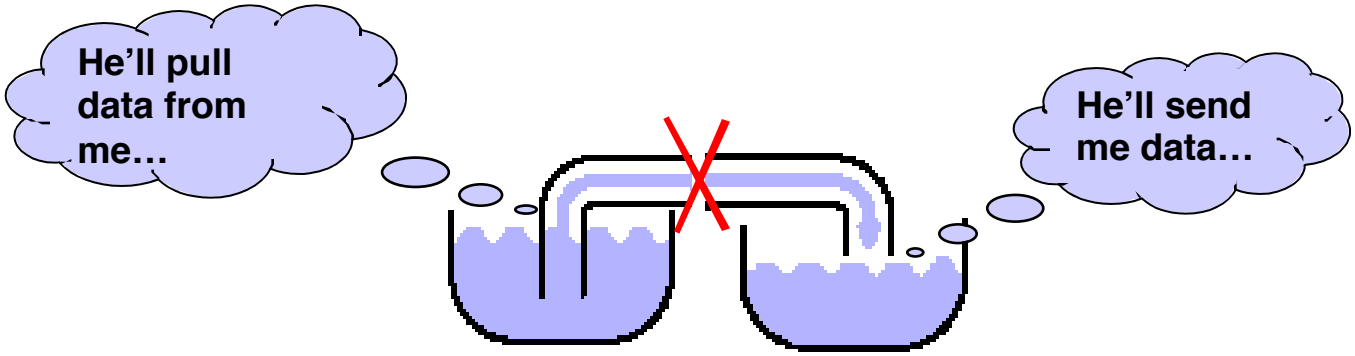
Alternatively …

- Infopipes can understand *push* messages



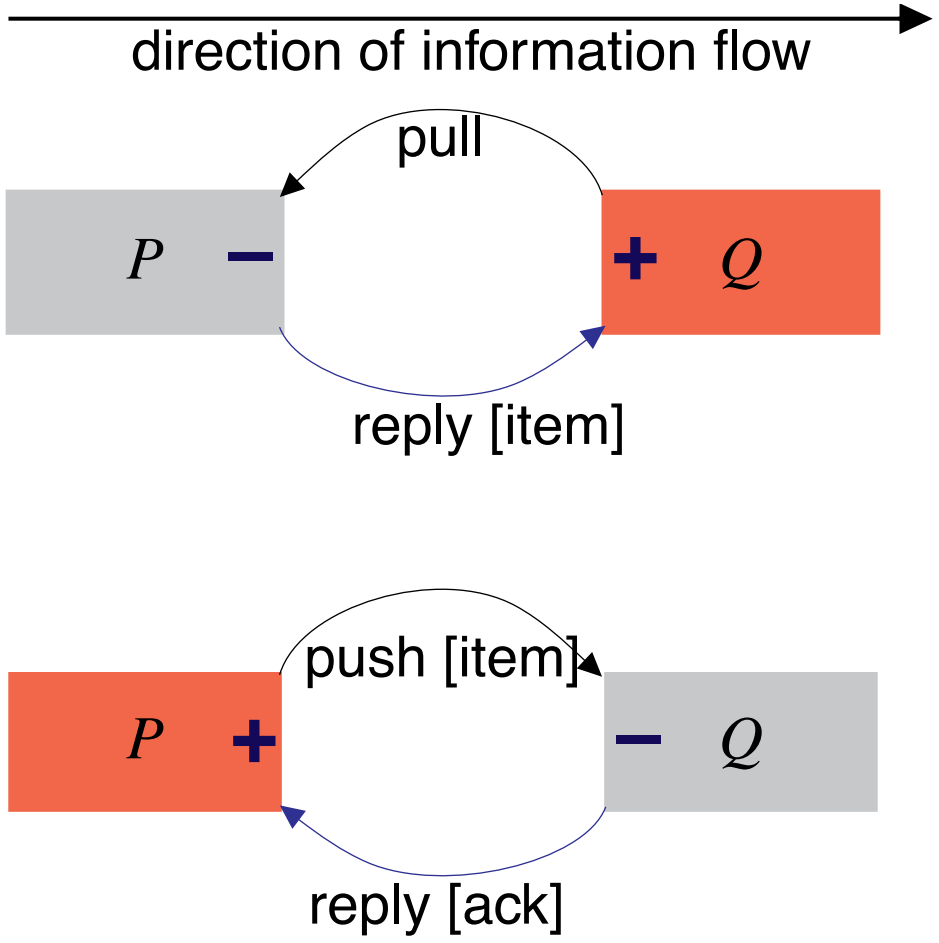
  - component *Q* waits for its upstream neighbour *P* to push an item into *Q*
- Both styles can be effective…
  - but one has to be careful when they are mixed

A Bad Connection



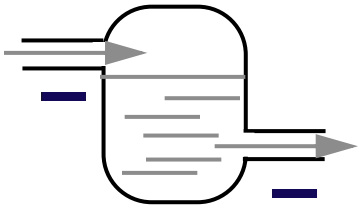
- We detect attempts to make bad connections with a *polarity* system
  - Each port has a polarity (+ or –)
  - as well as a direction (input or output)

Polarity


direction of information flow
pull
P
−
+
Q
reply [item]
push [item]
P
+
−
Q
reply [ack]

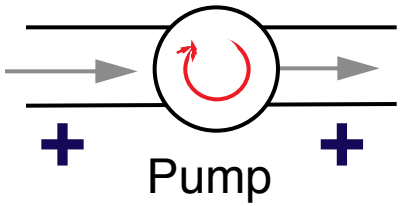
Buffers & Pumps

- Buffers are "naturally" negative at their inport and their outport.
  - Buffers "absorb energy" from the system (like *Lazy evaluation*)



Buffer

- Pumps are the dual; they are positive at both inport and outport.
  - Pumps "add energy" to the system (*drive the evaluation*)

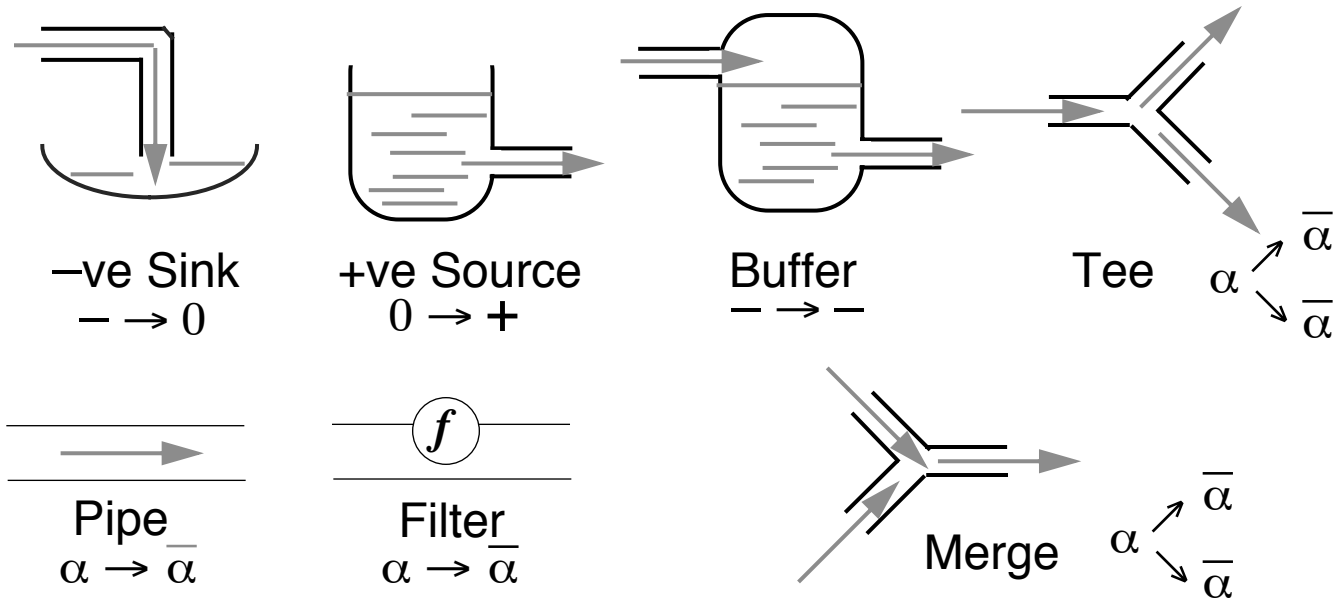


Pump

Polarity Checking

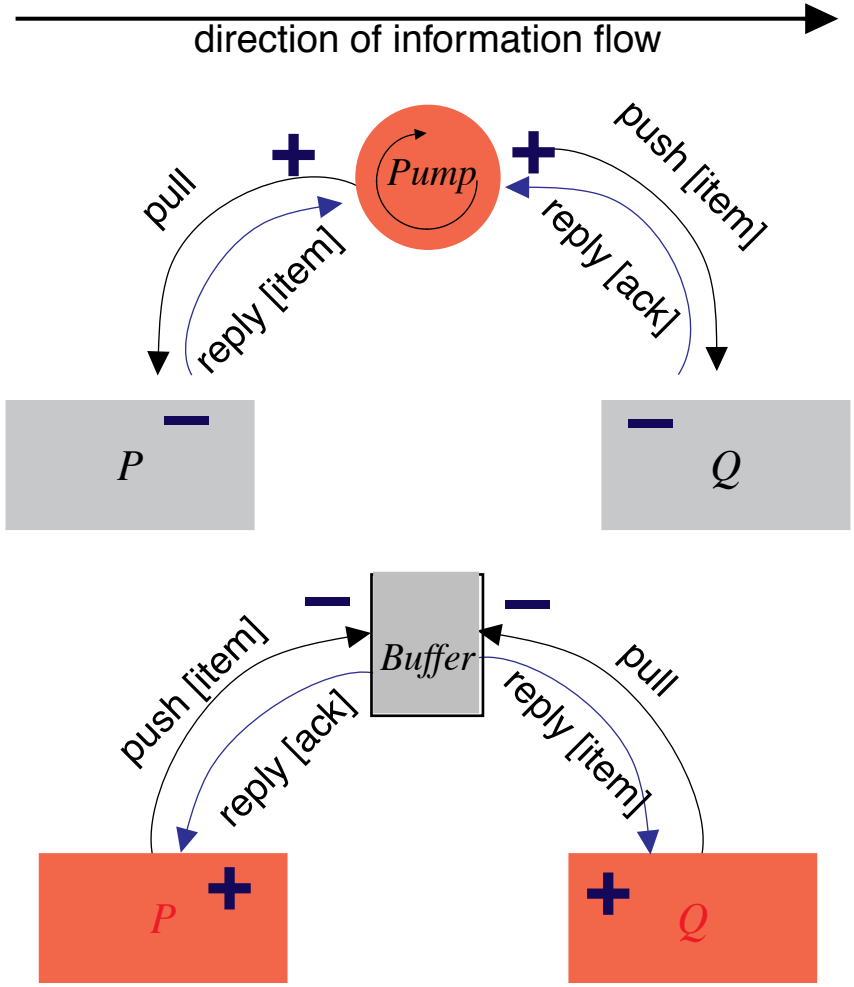
- In a well-formed pipeline, two pumps cannot be connected together.
  - nor can two buffers.
- Examine the polarity of the ports that are about to be connected:
  - unlike poles attract and discharge each other;
  - like poles repel; they cannot be connected.

Polarities of InfoPipe Components



- some components are “polymorphic”; they work equally well as + → − or − → +.
- others come in multiple varieties

Building Pipelines



Implementation Styles

- A component coded as – →+ will be quite different from one coded as + →–
  - … even though they perform the same function.
  - Example: defragmenter, taking two input items and assembing them into a composit item

```
Defragmenter ›› pull
   | item1 item2 |
   item1 := inport get.
   item2 := inport get.
   ↑ self
        assemble: item1
             and: item2.
```

```
Defragmenter ›› push: item
   isFirst
      ifTrue: [
         buffer := item]
      ifFalse: [
         outport put:
            (self
               assemble: buffer
                    and: item) ].
   isFirst := isFirst not.
```

Implementation Styles (cont.)

- There is also a third style, for a + → + degfragmentor.

```
Defragmenter ›› stroke
   | item1 item2 |
   item1 := inport get.
   item2 := inport get.
   outport put:
           (self
              assemble: item1
                   and: item2)
```

```
Defragmenter ›› startPumping:
                          period
   self strokeInterval: period.
   [[ self stroke.
      strokeDelay wait]
              repeatForever]fork
```

- What can we do to avoid re-writing code in all of these ways?

Transforming Implementations

- Use Infopipe composition ✓
  - don't transform the component!
  - compose it with pumps & buffers as required
- Use middleware ✓
  - Middleware libraries provide "clever" implementations for `inport get` and `outport put`
  - Use coroutines rather than threads for efficiency

Koster, Black, Huang, Walpole & Pu, *Middleware 2001*

Transforming Implementations (cont.)

- Transform the source code ✘
  - not feasible with conventional languages, *e.g.*, C++
- Use domain-specific source language ?
  - higher level semantics and limited expressiveness
  - may make it possible to generate code in whatever form is required
  - currently under investigation …

Status

- InfoPipe abstractions prototyped
  - Data and Connection interfaces defined
    - Polarity checker implemented
  - Particular Control Interfaces in use
  - Network MIDI player implemented
    - timely playout with minimal buffering
- Streaming video using real-time labeling
  - to iPac, from robot, on real-time OS
  - not yet re-implemented as Infopipes

Next Steps

- Investigating *blocking*.
  - What to do when pushing into a full buffer?
- Re-implement infopipes in a real-time framework
  - Learn more about control interfaces, *e.g.*
    - using real-time labels rather than buffer fill levels to provide rate feedback.
  - Deal with overload conditions, when active Infopipe components cannot get the resources that they need.

Are Objects the Right Abstraction?

- Why should the Infopipe programmer care about polarity?
  - The direction of data flow is all that matters
  - Does the placement of processes have to be visible?
- Can we design a higher-level *Domain Specific Language* from which
  - Placement of pumps (processes) can be inferred?
  - Objects can be synthesized in whatever polarity is needed?

Related Work

- Programming Model
  - Flow- based programming (Morrison)
- Components and configuration
  - Regis (Imperial College), QoS Dream (U Cambridge)
- Streams
  - CORBA, TAO (Washington U), MULTE (U Oslo)
- Protocol frameworks
  - Ensemble (Cornell), x-kernel + Scout (U Arizona)
- Multimedia frameworks
  - VuSystem (MIT), Mash (Berkeley) GStreamer (free), DirectShow (Microsoft)