Teleport Messaging for Distributed Stream Programs

William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah and Saman Amarasinghe

Massachusetts Institute of Technology

PPoPP 2005

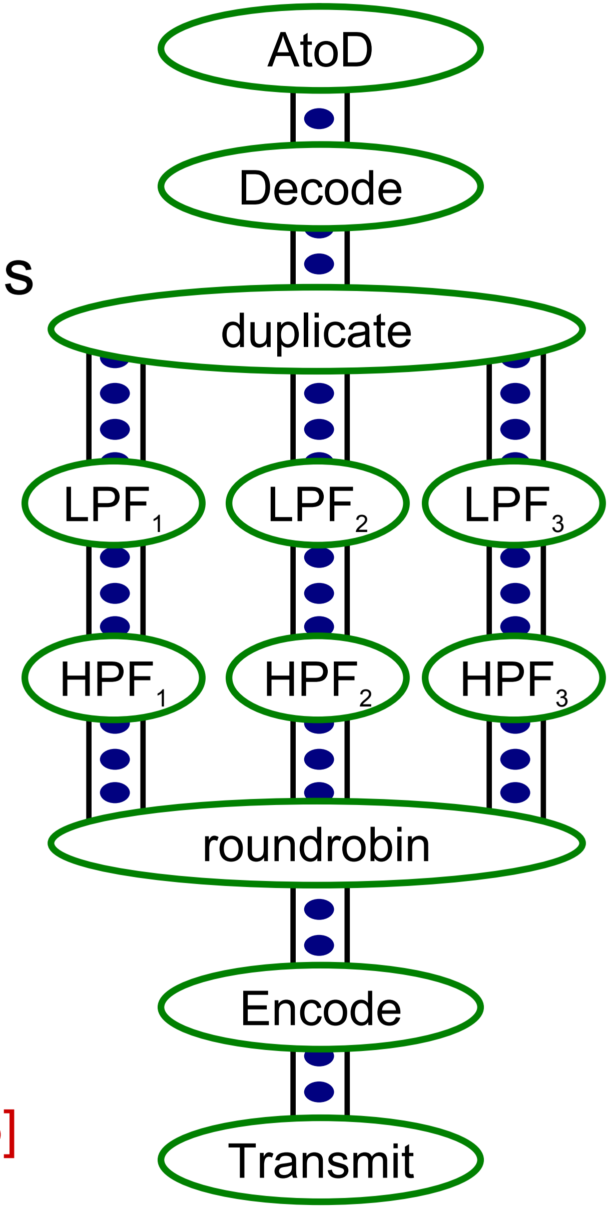
http://cag.lcs.mit.edu/streamit

Streaming Application Domain

- Based on a stream of data
  - Radar tracking, microphone arrays, HDTV editing, cell phone base stations
  - Graphics, multimedia, software radio
- Properties of stream programs
  - Regular and repeating computation
  - Parallel, independent actors with explicit communication
  - Data items have short lifetimes

➡ Amenable to aggressive compiler optimization

[ASPLOS '02, PLDI '03, LCTES'03, LCTES '05]

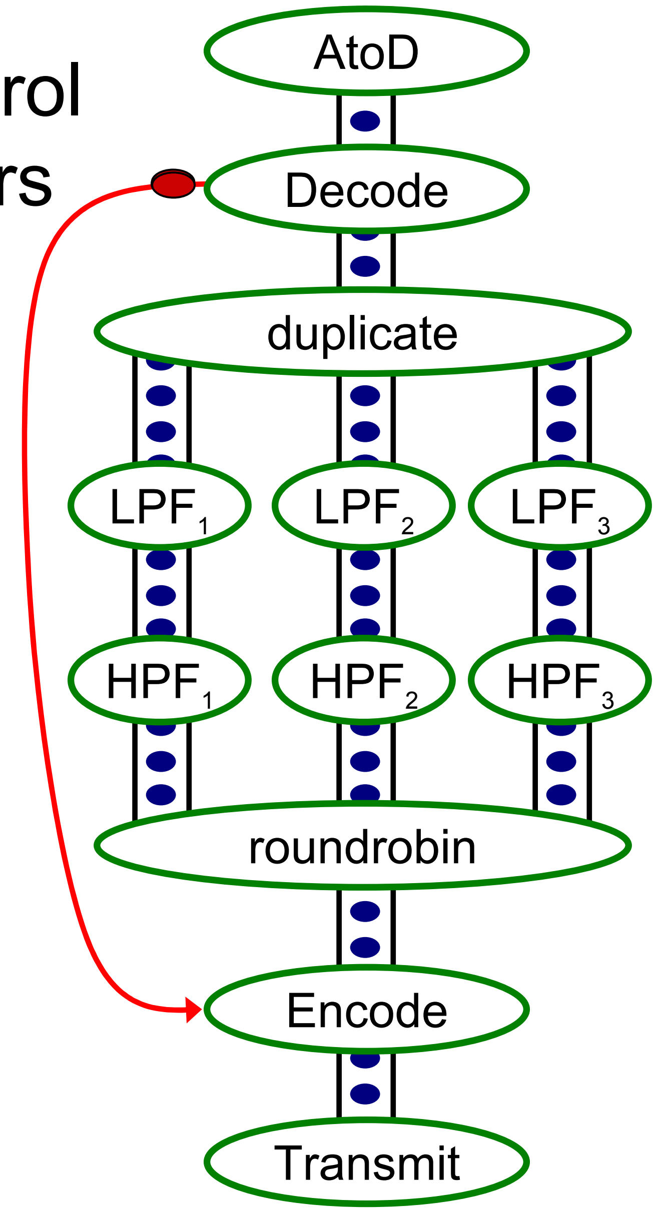


Control Messages

- Occasionally, low-bandwidth control messages are sent between actors
- Often demands precise timing
  - Communications: adjust protocol, amplification, compression
  - Network router: cancel invalid packet
  - Adaptive beamformer: track a target
  - Respond to user input, runtime errors
  - Frequency hopping radio

➡ What is the right programming model?

➡ How to implement efficiently?



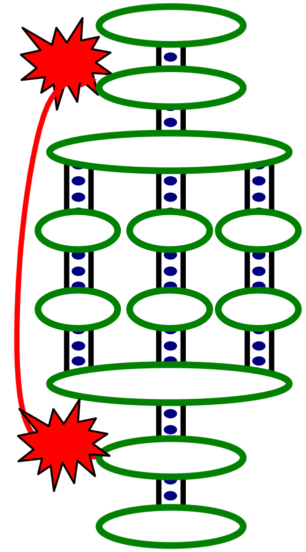
Supporting Control Messages

- Option 1: Synchronous method call

  PRO: - delivery transparent to user

  CON: - timing is unclear

  - limits parallelism



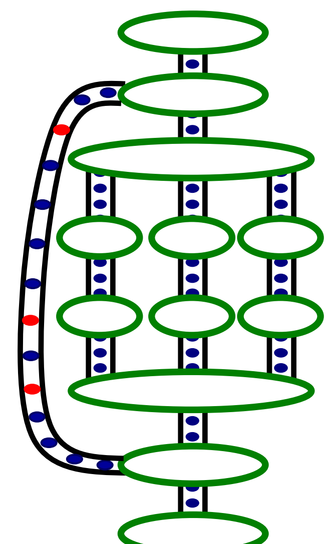
- Option 2: Embed message in stream

  PRO: - message arrives with data

  CON: - complicates filter code

  - complicates stream graph

  - runtime overhead

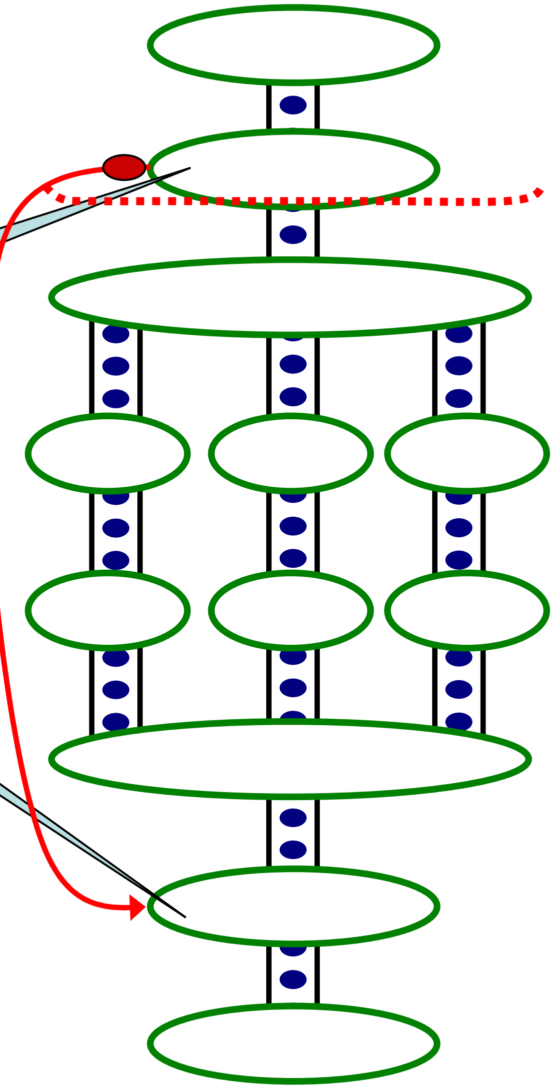


Teleport Messaging

- Looks like method call, but timed relative to data in the stream

```
TargetFilter x;
if newProtocol(p) {
  x.setProtocol(p) @ 2;
}
```

```
void setProtocol(int p) {
  reconfig(p);
}
```



- PRO:
  - simple and precise for user
    - adjustable latency
    - can send upstream or downstream
  - exposes dependences to compiler

Outline

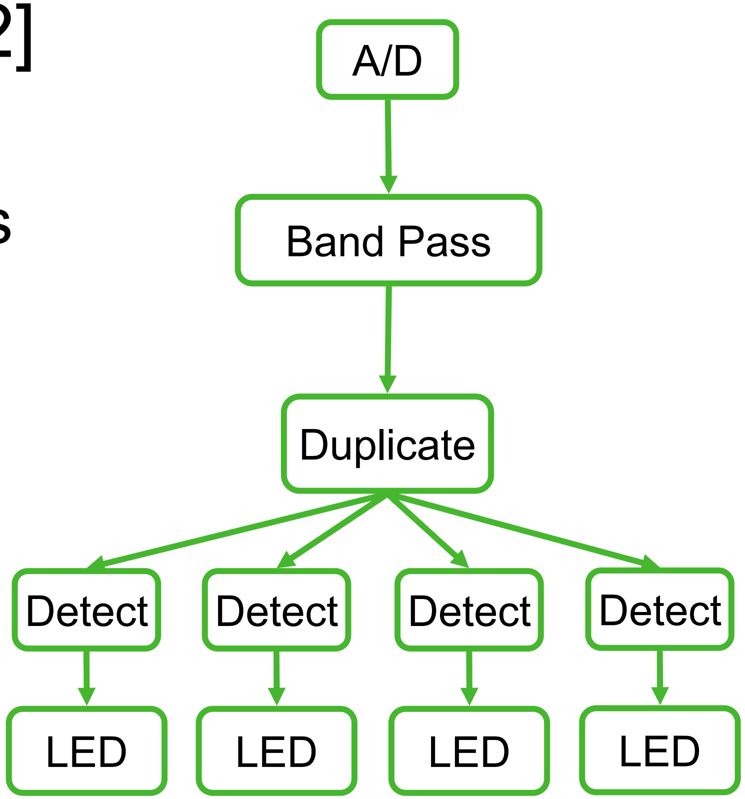
- StreamIt
- Teleport Messaging
- Case Study
- Related Work and Conclusion

Outline

- StreamIt
- Teleport Messaging
- Case Study
- Related Work and Conclusion

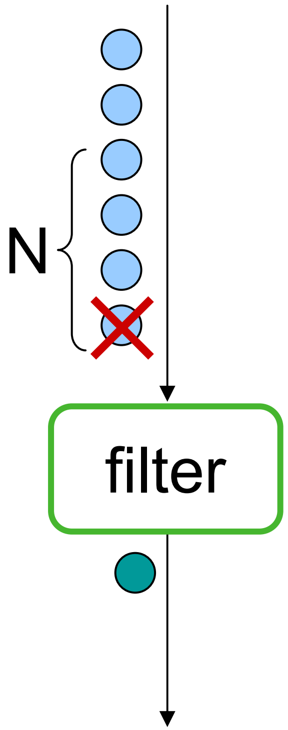
Model of Computation

- Synchronous Dataflow [Lee 92]
  - Graph of autonomous filters
  - Communicate via FIFO channels
  - Static I/O rates
- Compiler decides on an order of execution (schedule)
  - Many legal schedules



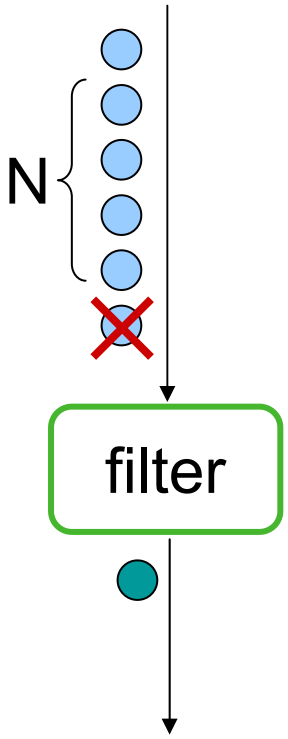
Example StreamIt Filter

```
float->float filter LowPassFilter (int N, float[N] weights) {
  work peek N push 1 pop 1 {
    float result = 0;
    for (int i=0; i<weights.length; i++) {
      result += weights[i] * peek(i);
    }
    push(result);
    pop();
  }
}
```



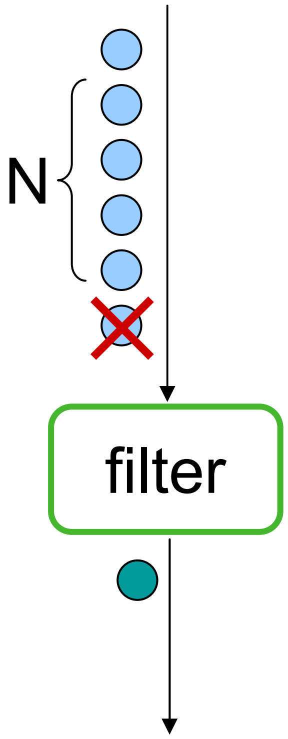
Example StreamIt Filter

```
float->float filter LowPassFilter (int N, float[N] weights) {
    work peek N push 1 pop 1 {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }

    handler setWeights(float[N] _weights) {
        weights = _weights;
    }
}
```



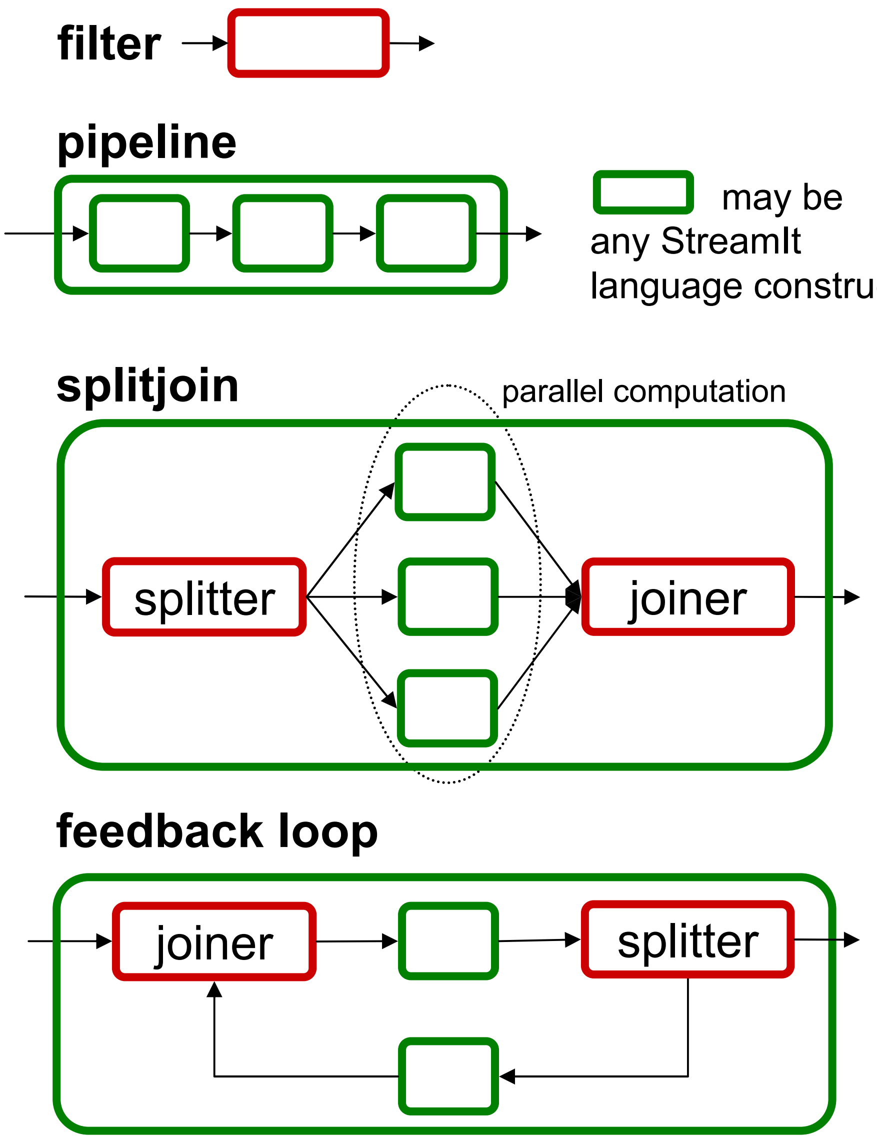
Example StreamIt Filter

```
float->float filter LowPassFilter (int N, float[N] weights, Frontend f) {
    work peek N push 1 pop 1 {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        if (result == 0) {
            f.increaseGain() @ [2:5];
        }
        push(result);
        pop();
    }

    handler setWeights(float[N] _weights) {
        weights = _weights;
    }
}
```



StreamIt Language Overview

- StreamIt is a novel language for streaming
  - Exposes parallelism and communication
  - Architecture independent
  - Modular and composable
    - Simple structures composed to creates complex graphs
  - Malleable
    - Change program behavior with small modifications

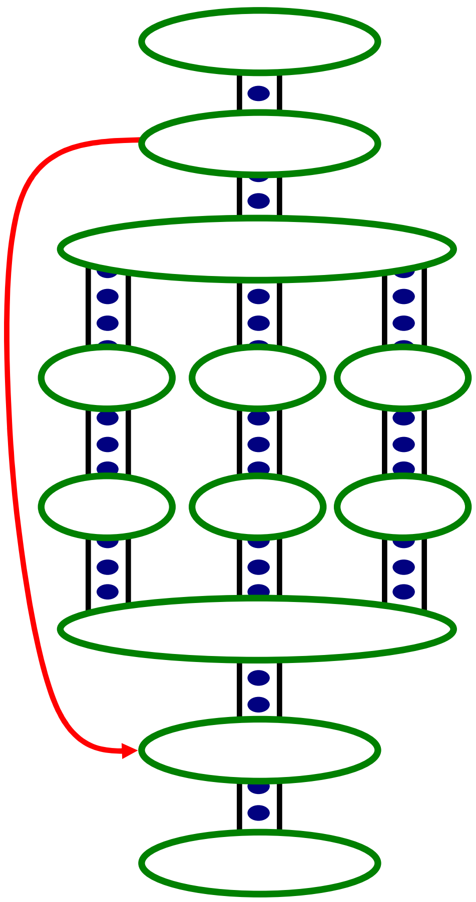


Outline

- StreamIt
- Teleport Messaging
- Case Study
- Related Work and Conclusion

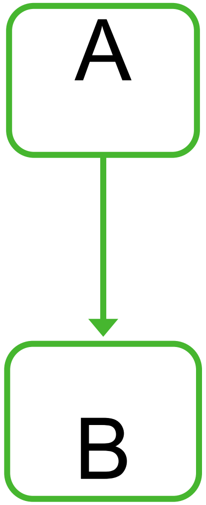
Providing a Common Timeframe

- Control messages need precise timing with respect to data stream
- However, there is no global clock in distributed systems
  - Filters execute independently, whenever input is available
- Idea: define message timing with respect to data dependences
  - Must be robust to multiple datarates
  - Must be robust to splitting, joining



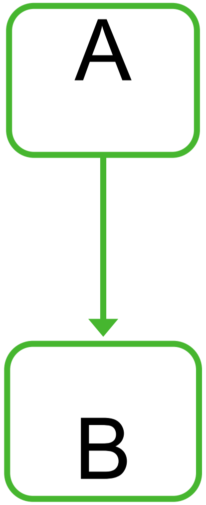
Stream Dependence Function (SDEP)

- Describes data dependences between filters



Stream Dependence Function (SDEP)

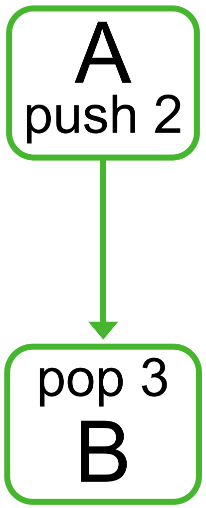
- Describes data dependences between filters



$\text{SDEP}_{\mathbf{A \leftarrow B}}(\mathbf{n})$: minimum number of times that **A** must execute to make it possible for **B** to execute **n** times

Stream Dependence Function (SDEP)

- Describes data dependences between filters

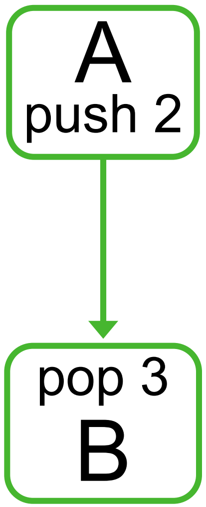


n	$SDEP_{\mathbf{A \leftarrow B}}(\mathbf{n})$
0	
1	
2	

$SDEP_{\mathbf{A \leftarrow B}}(\mathbf{n})$: minimum number of times that **A** must execute to make it possible for **B** to execute **n** times

Stream Dependence Function (SDEP)

- Describes data dependences between filters

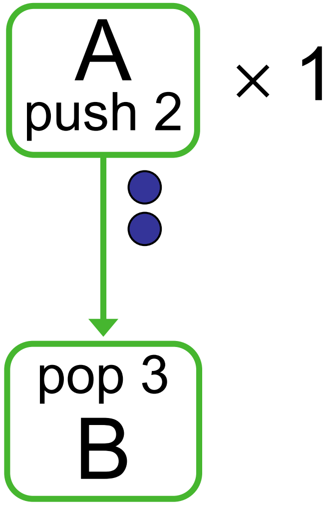


n	$\mathrm{SDEP}_{\mathbf{A} \leftarrow \mathbf{B}}(\mathbf{n})$
0	0
1	
2	

$\mathrm{SDEP}_{\mathbf{A} \leftarrow \mathbf{B}}(\mathbf{n})$: minimum number of times that **A** must execute to make it possible for **B** to execute **n** times

Stream Dependence Function (SDEP)

- Describes data dependences between filters

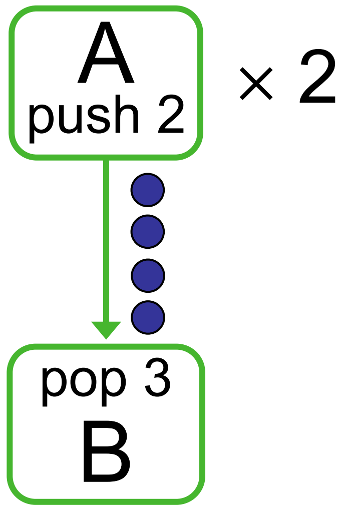


n	$SDEP_{A \leftarrow B}(\mathbf{n})$
0	0
1	
2	

$SDEP_{A \leftarrow B}(\mathbf{n})$: minimum number of times that **A** must execute to make it possible for **B** to execute **n** times

Stream Dependence Function (SDEP)

- Describes data dependences between filters

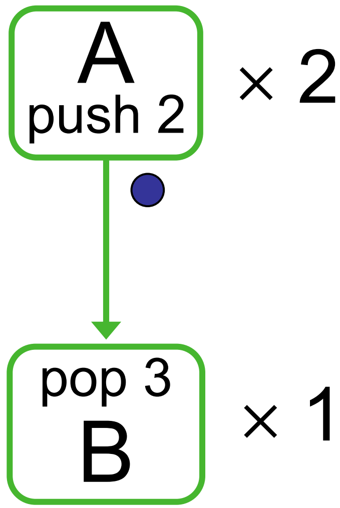


n	$\mathrm{SDEP}_{\mathbf{A} \leftarrow \mathbf{B}}(\mathbf{n})$
0	0
1	
2	

$\mathrm{SDEP}_{\mathbf{A} \leftarrow \mathbf{B}}(\mathbf{n})$: minimum number of times that **A** must execute to make it possible for **B** to execute **n** times

Stream Dependence Function (SDEP)

- Describes data dependences between filters

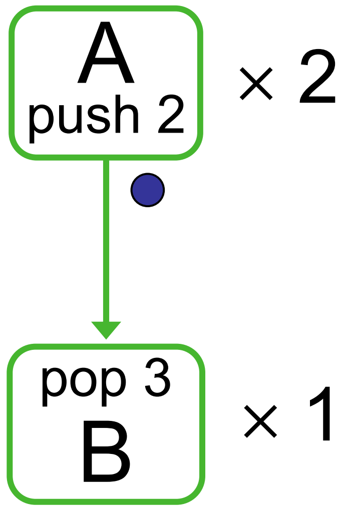


n	$\mathrm{SDEP}_{A \leftarrow B}(\mathbf{n})$
0	0
1	
2	

$\mathrm{SDEP}_{A \leftarrow B}(\mathbf{n})$: minimum number of times that **A** must execute to make it possible for **B** to execute **n** times

Stream Dependence Function (SDEP)

- Describes data dependences between filters

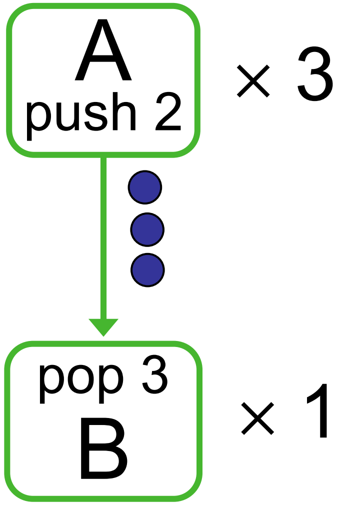


n	$\text{SDEP}_{\mathbf{A \leftarrow B}}(\mathbf{n})$
0	0
1	2
2	

$\text{SDEP}_{\mathbf{A \leftarrow B}}(\mathbf{n})$: minimum number of times that **A** must execute to make it possible for **B** to execute **n** times

Stream Dependence Function (SDEP)

- Describes data dependences between filters

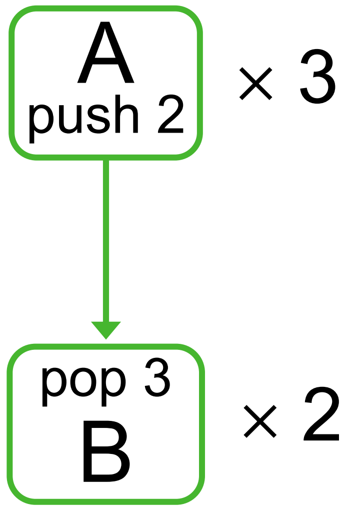


n	$SDEP_{A \leftarrow B}(\mathbf{n})$
0	0
1	2
2	

$SDEP_{A \leftarrow B}(\mathbf{n})$: minimum number of times that **A** must execute to make it possible for **B** to execute **n** times

Stream Dependence Function (SDEP)

- Describes data dependences between filters

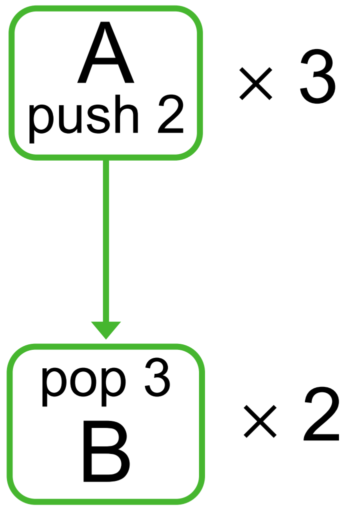


n	$SDEP_{A \leftarrow B}(\mathbf{n})$
0	0
1	2
2	

$SDEP_{A \leftarrow B}(\mathbf{n})$: minimum number of times that **A** must execute to make it possible for **B** to execute **n** times

Stream Dependence Function (SDEP)

- Describes data dependences between filters

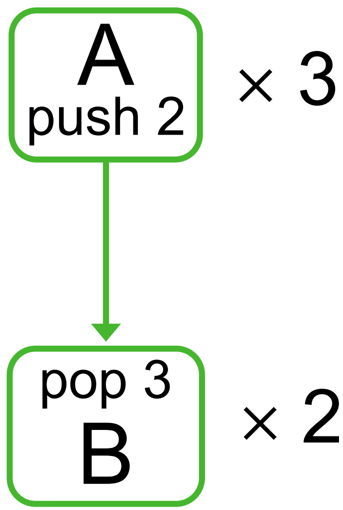


n	$SDEP_{\mathbf{A \leftarrow B}}(\mathbf{n})$
0	0
1	2
2	3

$SDEP_{\mathbf{A \leftarrow B}}(\mathbf{n})$: minimum number of times that **A** must execute to make it possible for **B** to execute **n** times

Stream Dependence Function (SDEP)

- Describes data dependences between filters

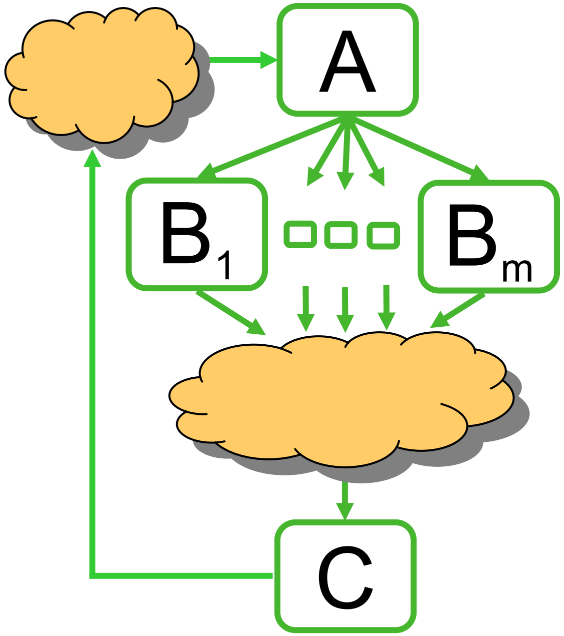


n	$SDEP_{A \leftarrow B}(\mathbf{n})$
0	0
1	2
2	3

$$= \left\lceil \frac{n*3}{2} \right\rceil$$

$SDEP_{A \leftarrow B}(\mathbf{n})$: minimum number of times that **A** must execute to make it possible for **B** to execute **n** times

Calculating SDEP: General Case



$$SDEP_{\mathbf{A \leftarrow C}}(n) = \max_{i \in [1,m]} [SDEP_{\mathbf{A \leftarrow Bi}}(SDEP_{\mathbf{Bi \leftarrow C}}(n))]$$

➡ SDEP is compositional

$SDEP_{\mathbf{A \leftarrow B}}(\mathbf{n})$: minimum number of times that **A** must execute to make it possible for **B** to execute **n** times

Teleport Messaging using SDEP

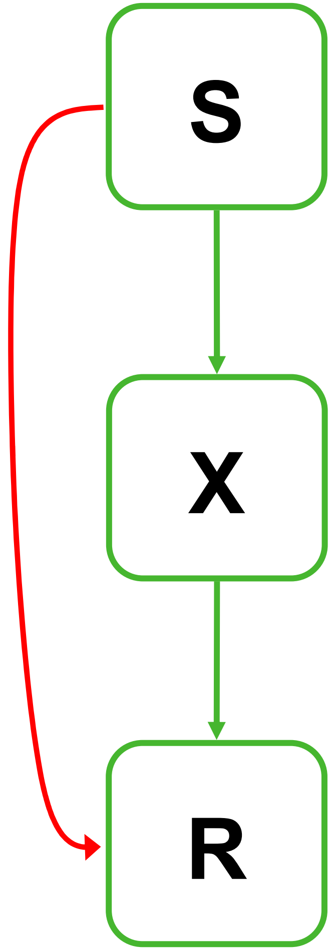
- SDEP provides precise semantics for message timing

If **S** sends message to **R**:

- on the **n**th execution of **S**
- with latency range [$\mathbf{k_1}$, $\mathbf{k_2}$]

Then message is delivered to **R**:

- on any iteration **m** such that
  $\mathbf{n+k_1} \leq \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \leq \mathbf{n+k_2}$



Teleport Messaging using SDEP

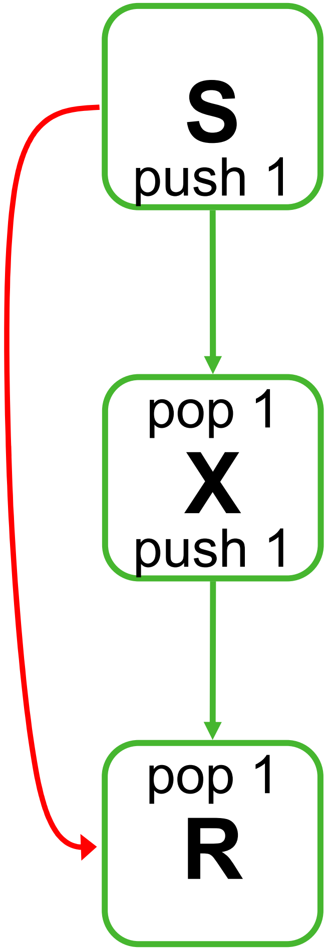
- SDEP provides precise semantics for message timing

If **S** sends message to **R**:

- on the **n**th execution of **S**
- with latency range [$\mathbf{k_1}$, $\mathbf{k_2}$]

Then message is delivered to **R**:

- on any iteration **m** such that $\mathbf{n+k_1} \leq \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \leq \mathbf{n+k_2}$



Teleport Messaging using SDEP

- SDEP provides precise semantics for message timing

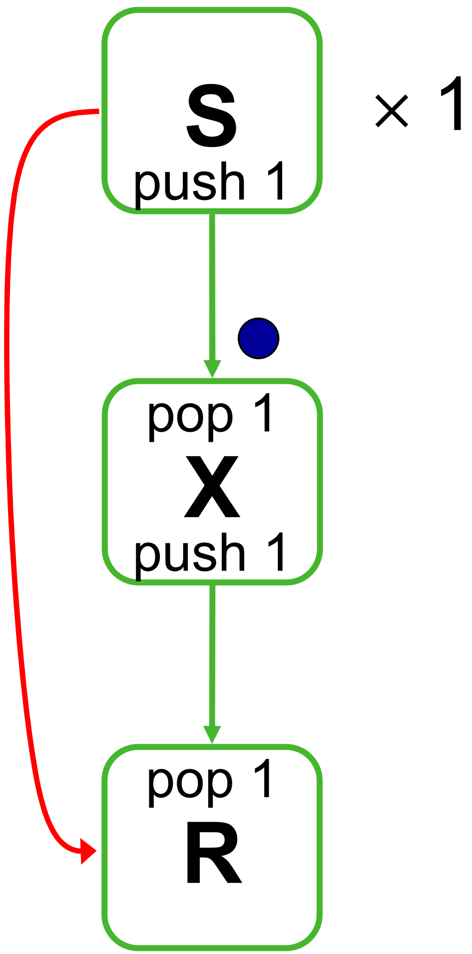
If **S** sends message to **R**:

- on the **n**th execution of **S**
- with latency range [$\mathbf{k_1}$, $\mathbf{k_2}$]

Then message is delivered to **R**:

- on any iteration **m** such that

  $\mathbf{n+k_1} \leq \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \leq \mathbf{n+k_2}$



Teleport Messaging using SDEP

- SDEP provides precise semantics for message timing

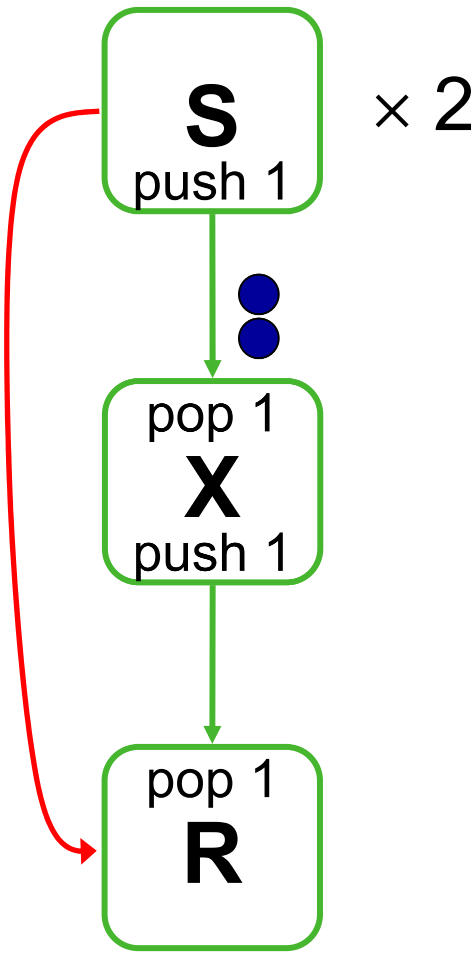
If **S** sends message to **R**:

- on the **n**th execution of **S**
- with latency range $[\mathbf{k_1}, \mathbf{k_2}]$

Then message is delivered to **R**:

- on any iteration **m** such that

  $\mathbf{n+k_1} \le \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \le \mathbf{n+k_2}$



Teleport Messaging using SDEP

- SDEP provides precise semantics for message timing

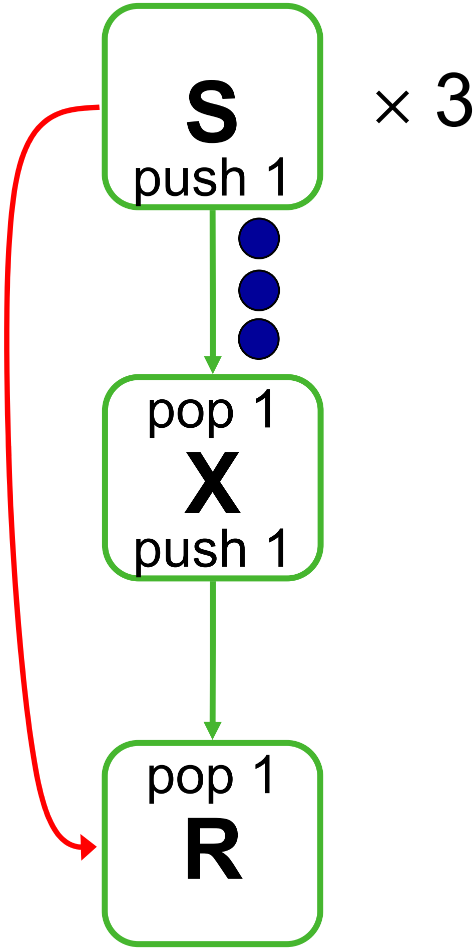
If **S** sends message to **R**:

- on the **n**th execution of **S**
- with latency range [$\mathbf{k_1}$, $\mathbf{k_2}$]

Then message is delivered to **R**:

- on any iteration **m** such that

  $\mathbf{n+k_1} \leq \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \leq \mathbf{n+k_2}$



Teleport Messaging using SDEP

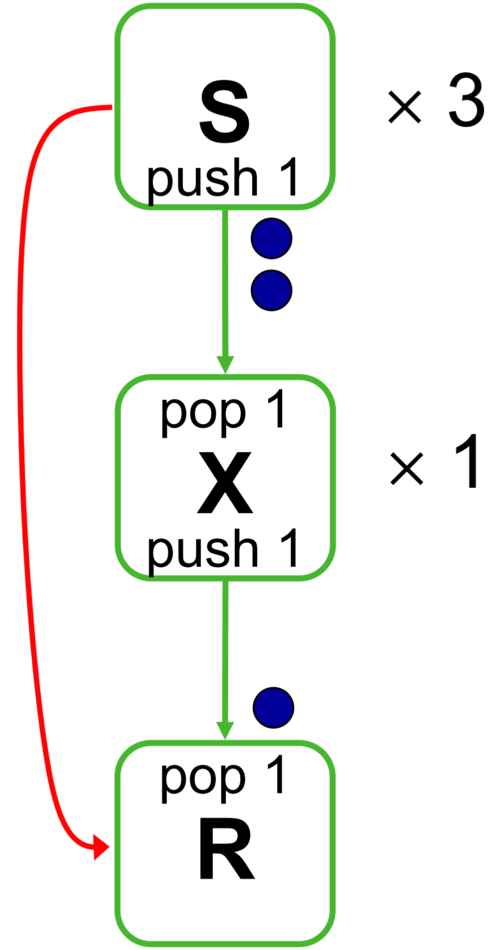
- SDEP provides precise semantics for message timing

If **S** sends message to **R**:

- on the **n**th execution of **S**
- with latency range $[\mathbf{k_1}, \mathbf{k_2}]$

Then message is delivered to **R**:

- on any iteration **m** such that
  $\mathbf{n+k_1} \leq \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \leq \mathbf{n+k_2}$



Teleport Messaging using SDEP

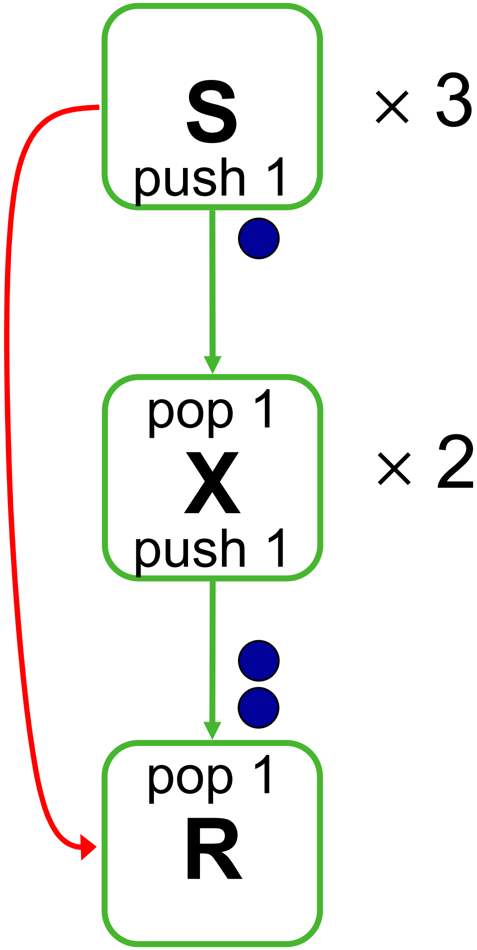
- SDEP provides precise semantics for message timing

If **S** sends message to **R**:

- on the **n**th execution of **S**
- with latency range [$\mathbf{k_1}$, $\mathbf{k_2}$]

Then message is delivered to **R**:

- on any iteration **m** such that $\mathbf{n+k_1} \leq \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \leq \mathbf{n+k_2}$



Teleport Messaging using SDEP

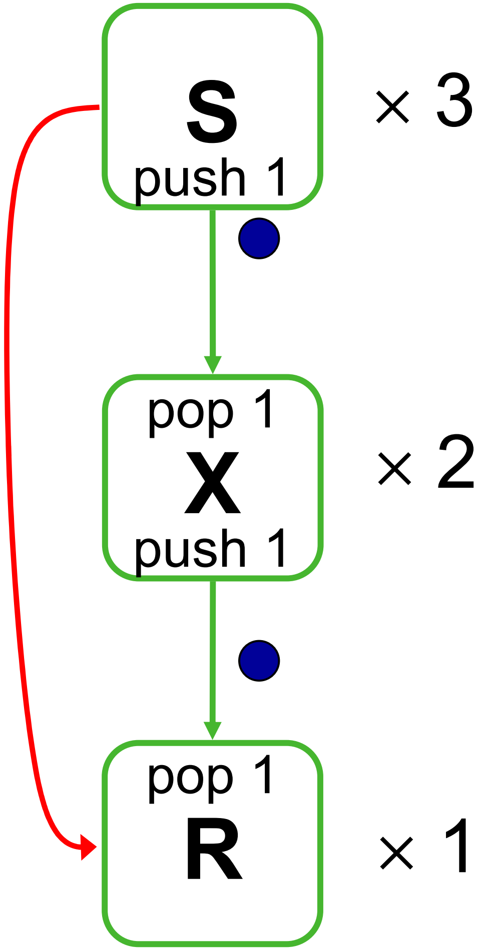
- SDEP provides precise semantics for message timing

If **S** sends message to **R**:

- on the **n**th execution of **S**
- with latency range [$\mathbf{k_1}$, $\mathbf{k_2}$]

Then message is delivered to **R**:

- on any iteration **m** such that
  $\mathbf{n+k_1} \leq \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \leq \mathbf{n+k_2}$



Teleport Messaging using SDEP

- SDEP provides precise semantics for message timing

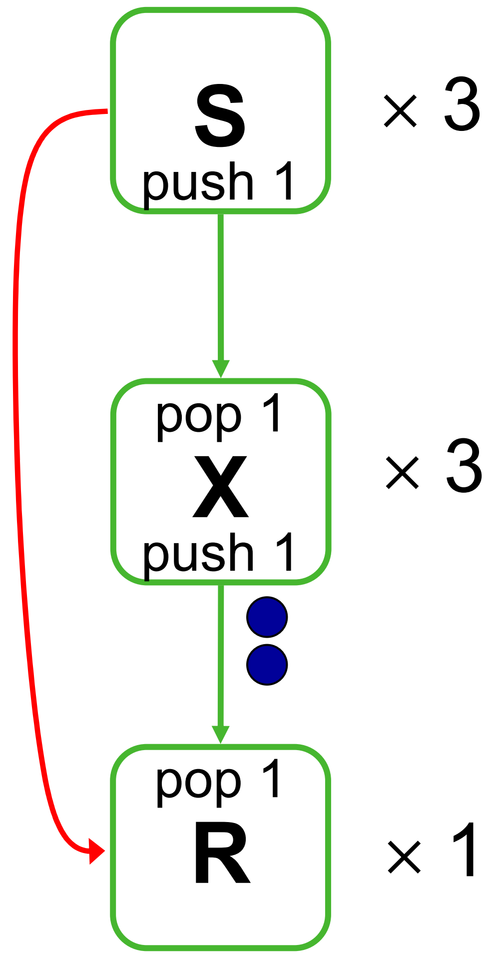
If **S** sends message to **R**:

- on the **n**th execution of **S**
- with latency range [$\mathbf{k_1}$, $\mathbf{k_2}$]

Then message is delivered to **R**:

- on any iteration **m** such that

  $\mathbf{n+k_1} \leq \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \leq \mathbf{n+k_2}$



Teleport Messaging using SDEP

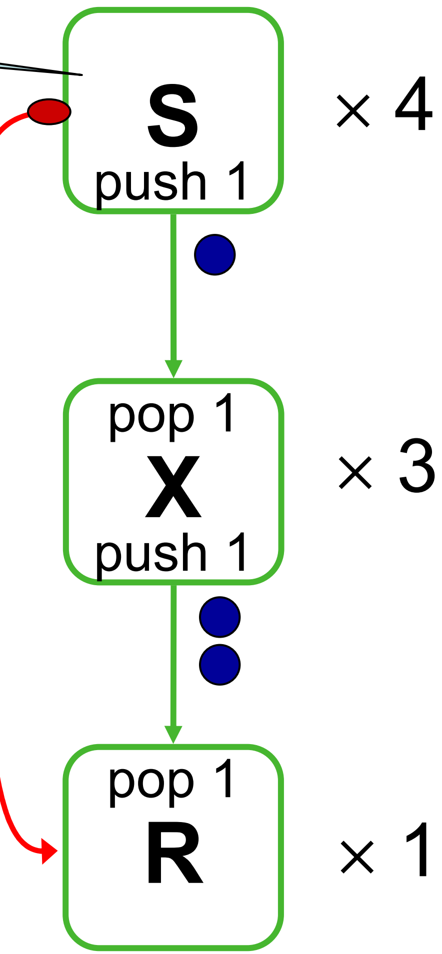
Receiver r;
r.increaseGain() @ [0:0]

If **S** sends message to **R**:

- on the **n**th execution of **S**
- with latency range [$\mathbf{k_1}$, $\mathbf{k_2}$]

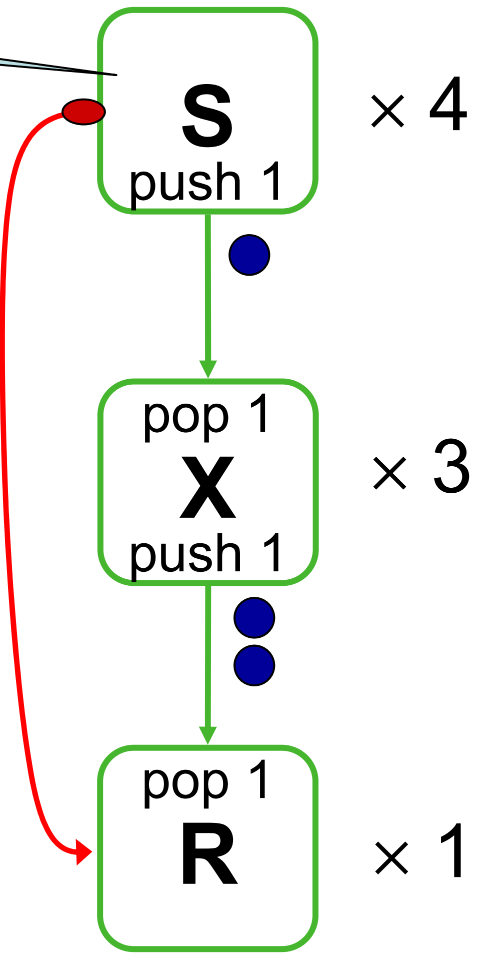
Then message is delivered to **R**:

- on any iteration **m** such that
  $\mathbf{n+k_1} \le \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \le \mathbf{n+k_2}$



Teleport Messaging using SDEP

Receiver r;
r.increaseGain() @ [0:0]



If **S** sends message to **R**:

- on the 4th execution of **S**
- with latency range [$\mathbf{k_1}$, $\mathbf{k_2}$]

Then message is delivered to **R**:

- on any iteration **m** such that

  $\mathbf{n+k_1} \leq \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \leq \mathbf{n+k_2}$

Teleport Messaging using SDEP

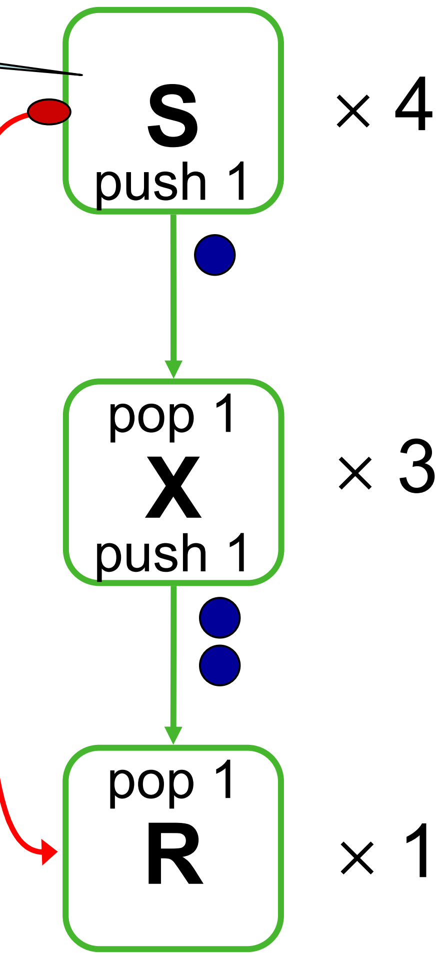
Receiver r;
r.increaseGain() @ [0:0]

If **S** sends message to **R**:

- on the **4**th execution of **S**
- with latency range **[0, 0]**

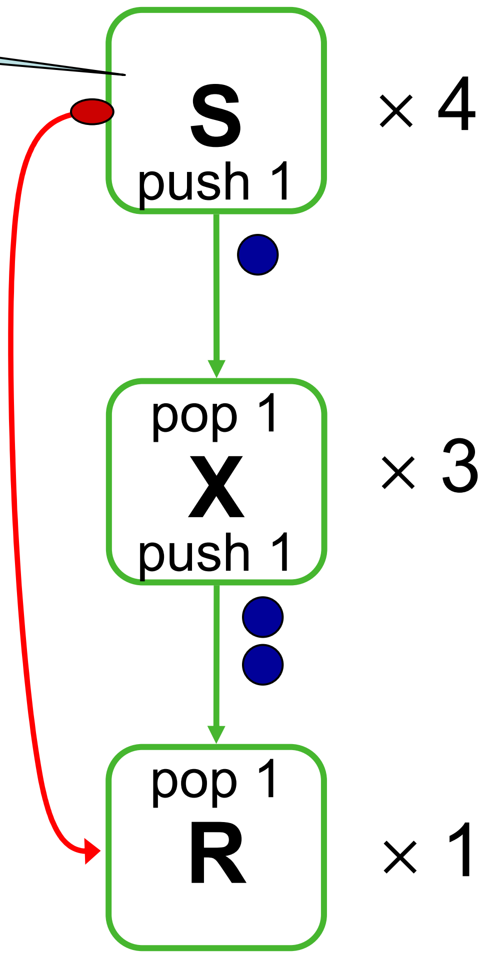
Then message is delivered to **R**:

- on any iteration **m** such that
  $\mathbf{n+k_1} \le \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \le \mathbf{n+k_2}$



Teleport Messaging using SDEP

Receiver r;
r.increaseGain() @ [0:0]



If **S** sends message to **R**:

- on the **4**th execution of **S**
- with latency range **[0, 0]**

Then message is delivered to **R**:

- on any iteration **m** such that

$\mathbf{4{+}0} \leq \mathrm{SDEP}_{\mathbf{S \leftarrow R}}(\mathbf{m}) \leq \mathbf{4{+}0}$

Teleport Messaging using SDEP

Receiver r;
r.increaseGain() @ [0:0]

If **S** sends message to **R**:

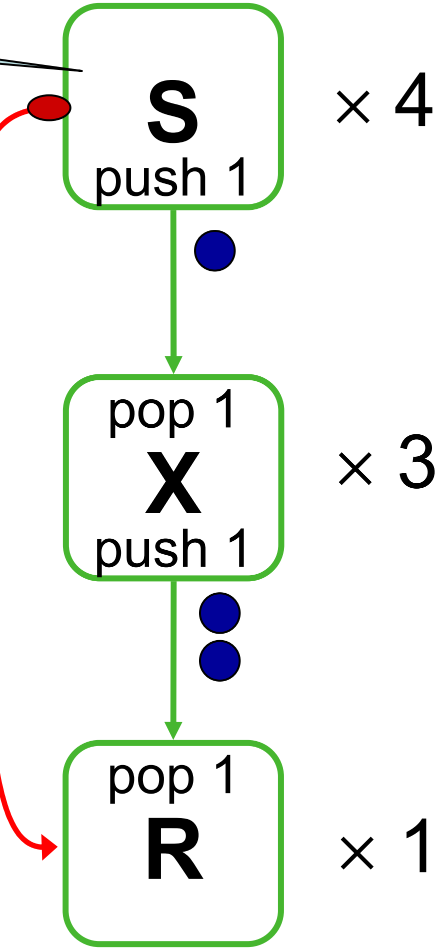
- on the **4th** execution of **S**
- with latency range **[0, 0]**

Then message is delivered to **R**:

- on any iteration **m** such that

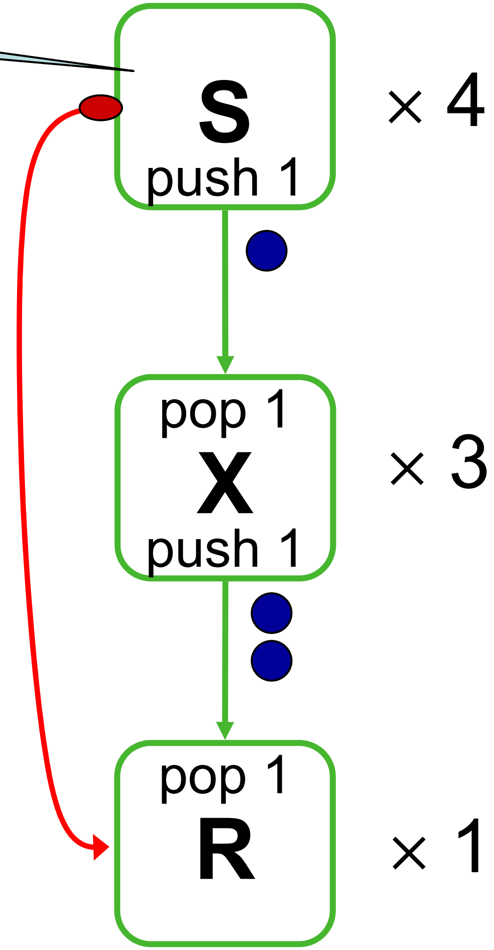
$$4+0 \le \mathrm{SDEP}_{S \leftarrow R}(\mathbf{m}) \le 4+0$$

$$\mathrm{SDEP}_{S \leftarrow R}(\mathbf{m}) = 4$$



Teleport Messaging using SDEP


```
Receiver r;
r.increaseGain() @ [0:0]
```



If **S** sends message to **R**:

- on the **4**th execution of **S**
- with latency range **[0, 0]**

Then message is delivered to **R**:

- on any iteration **m** such that

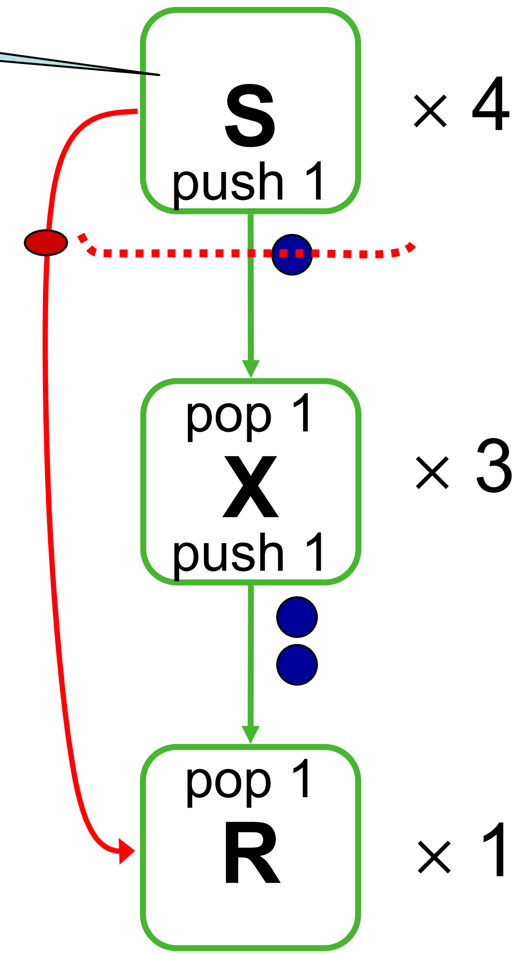
$$\mathbf{4+0} \leq \mathrm{SDEP}_{\mathbf{S\leftarrow R}}(\mathbf{m}) \leq \mathbf{4+0}$$

$$\mathrm{SDEP}_{\mathbf{S\leftarrow R}}(\mathbf{m}) = \mathbf{4}$$

$$\mathbf{m} = \mathbf{4}$$

Teleport Messaging using SDEP

```
Receiver r;
r.increaseGain() @ [0:0]
```



If **S** sends message to **R**:

- on the 4th execution of **S**
- with latency range [0, 0]

Then message is delivered to **R**:

- on any iteration **m** such that

$$4+0 \le \mathrm{SDEP}_{S \leftarrow R}(\mathbf{m}) \le 4+0$$

$$\mathrm{SDEP}_{S \leftarrow R}(\mathbf{m}) = 4$$

$$\mathbf{m} = 4$$

Teleport Messaging using SDEP

Receiver r;
r.increaseGain() @ [0:0]

If **S** sends message to **R**:

- on the 4th execution of **S**
- with latency range [0, 0]

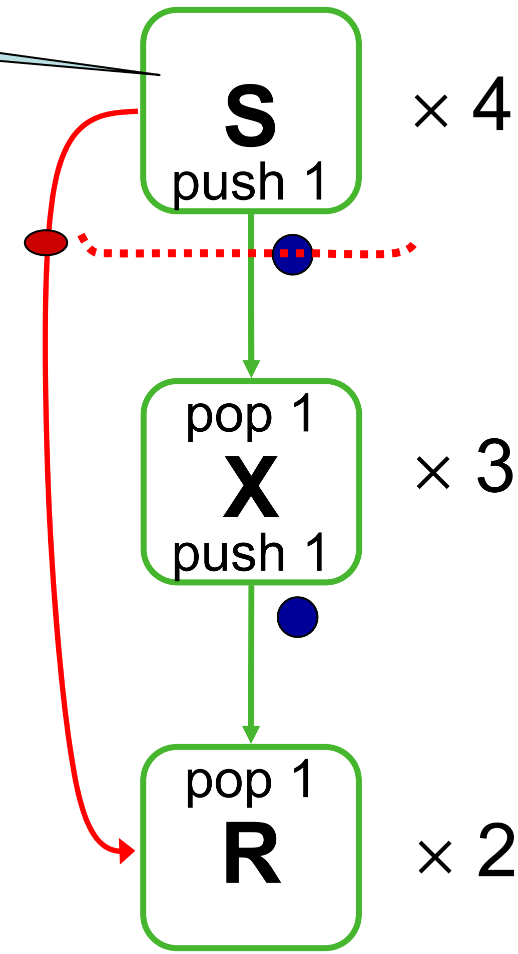
Then message is delivered to **R**:

- on any iteration **m** such that

$$4+0 \le \mathrm{SDEP}_{S \leftarrow R}(\mathbf{m}) \le 4+0$$

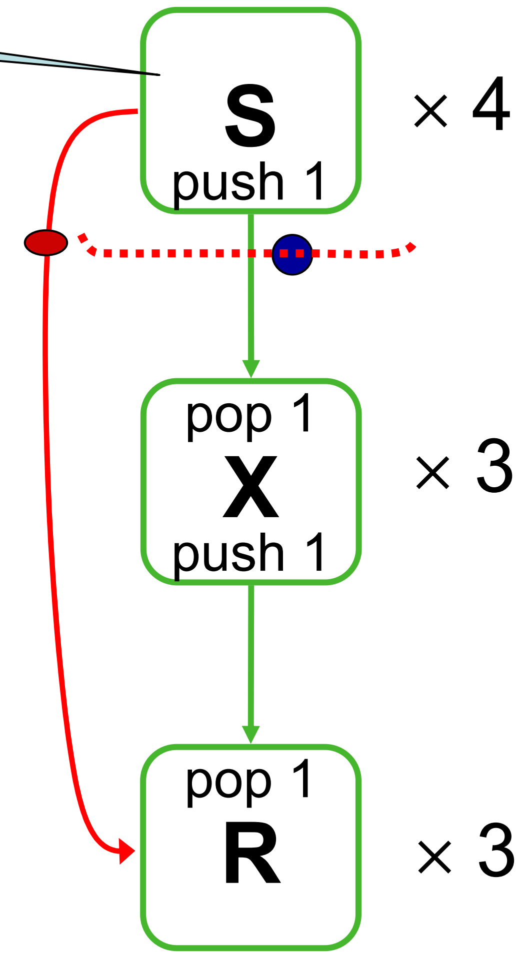
$$\mathrm{SDEP}_{S \leftarrow R}(\mathbf{m}) = 4$$

$$\mathbf{m} = 4$$



Teleport Messaging using SDEP

Receiver r;
r.increaseGain() @ [0:0]



If **S** sends message to **R**:

- on the **4th** execution of **S**
- with latency range **[0, 0]**

Then message is delivered to **R**:

- on any iteration **m** such that

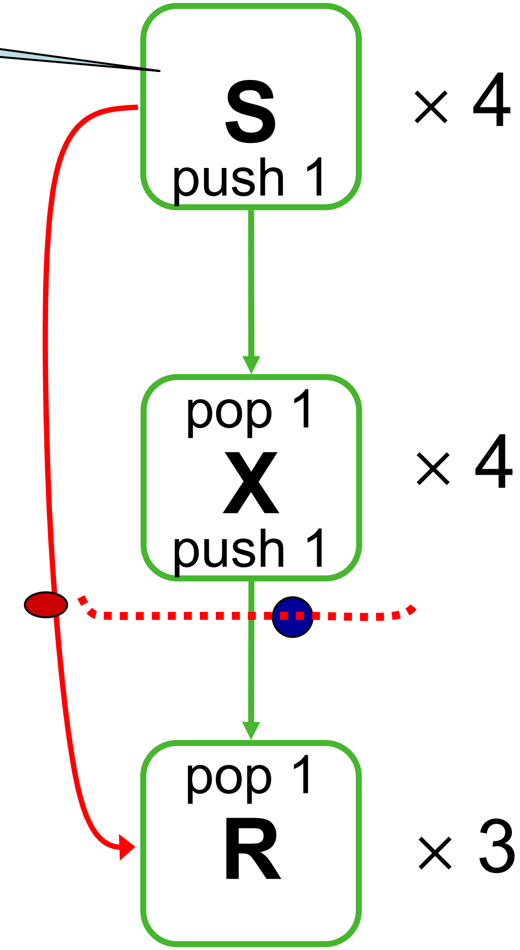
$$4+0 \leq \mathrm{SDEP}_{S \leftarrow R}(\mathbf{m}) \leq 4+0$$

$$\mathrm{SDEP}_{S \leftarrow R}(\mathbf{m}) = 4$$

$$\mathbf{m} = 4$$

Teleport Messaging using SDEP

Receiver r;
r.increaseGain() @ [0:0]



If **S** sends message to **R**:

- on the 4th execution of **S**
- with latency range [0, 0]

Then message is delivered to **R**:

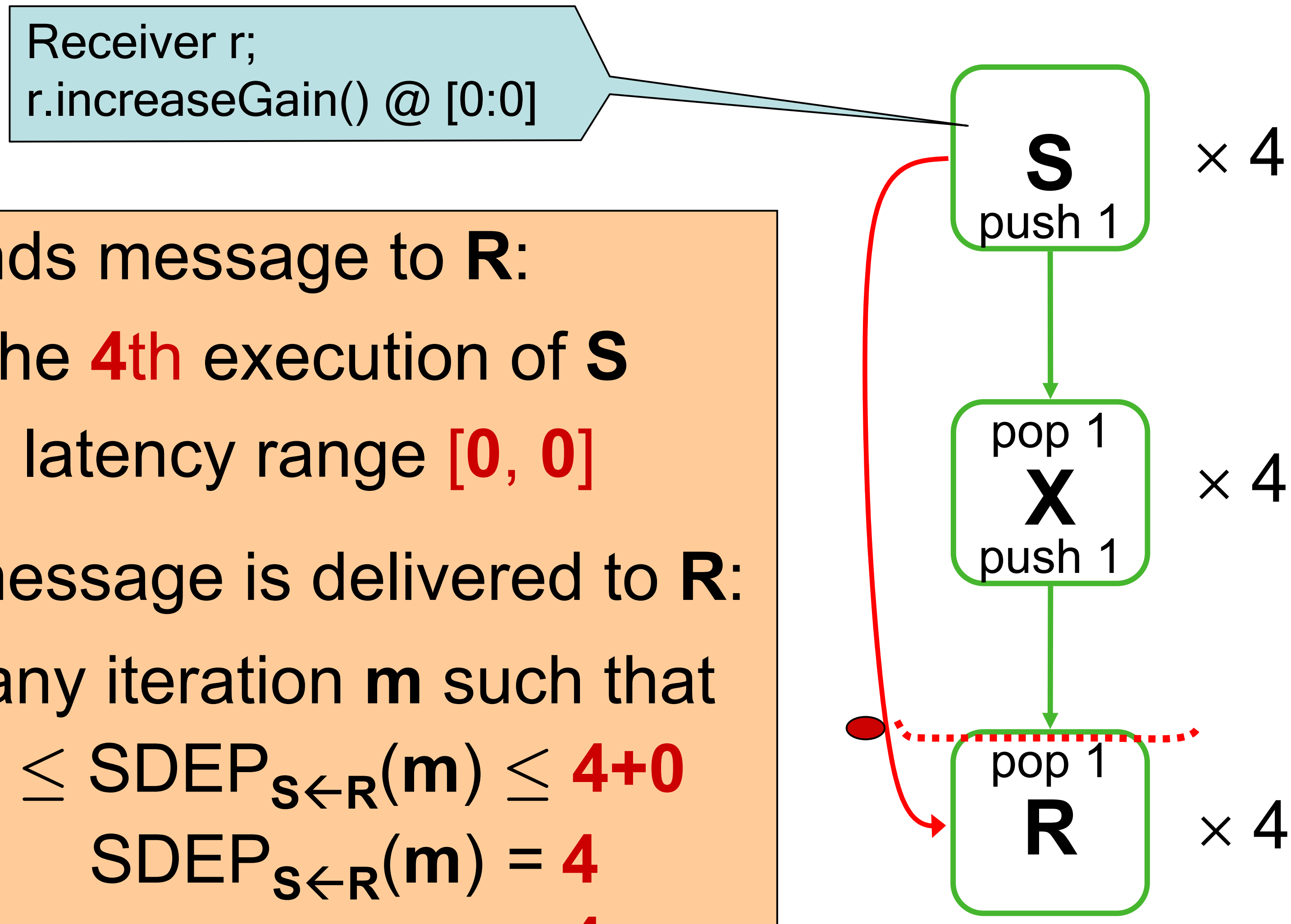
- on any iteration **m** such that

$$4+0 \le \mathrm{SDEP}_{S \leftarrow R}(\mathbf{m}) \le 4+0$$

$$\mathrm{SDEP}_{S \leftarrow R}(\mathbf{m}) = 4$$

$$\mathbf{m} = 4$$

Teleport Messaging using SDEP



If **S** sends message to **R**:

- on the 4th execution of **S**
- with latency range [0, 0]

Then message is delivered to **R**:

- on any iteration **m** such that

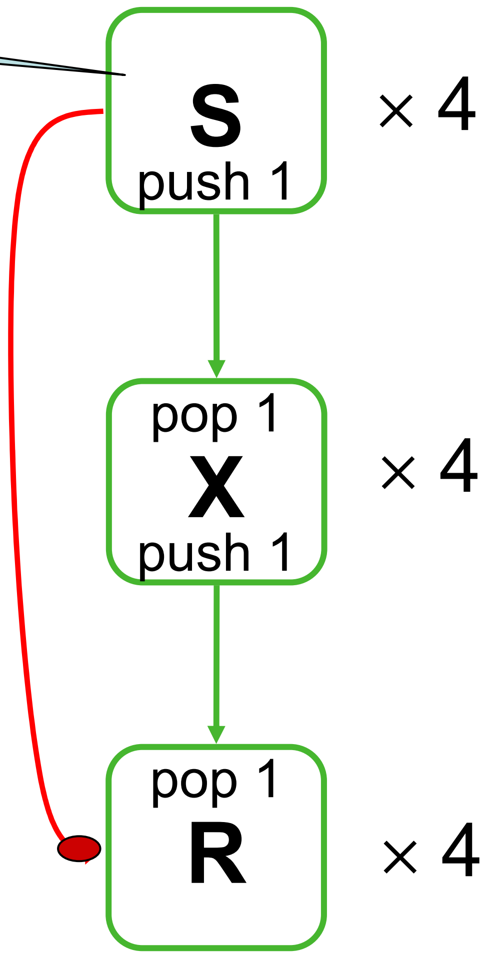
$$4+0 \leq \mathrm{SDEP}_{S \leftarrow R}(\mathbf{m}) \leq 4+0$$

$$\mathrm{SDEP}_{S \leftarrow R}(\mathbf{m}) = 4$$

$$\mathbf{m} = 4$$

Teleport Messaging using SDEP

Receiver r;
r.increaseGain() @ [0:0]



If **S** sends message to **R**:

- on the 4th execution of **S**
- with latency range [0, 0]

Then message is delivered to **R**:

- on any iteration **m** such that

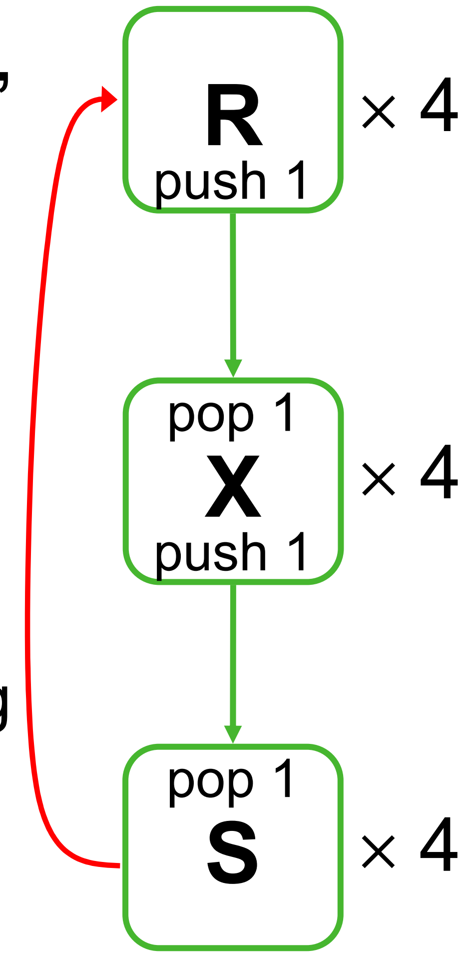
$$4+0 \leq \mathrm{SDEP}_{S \leftarrow R}(m) \leq 4+0$$

$$\mathrm{SDEP}_{S \leftarrow R}(m) = 4$$

$$m = 4$$

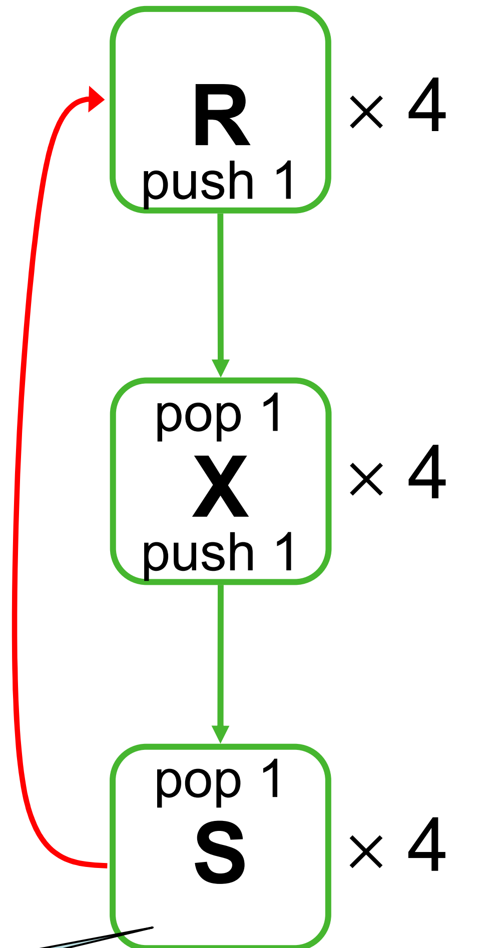
Sending Messages Upstream

- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides provides a unified abstraction
- Intuition:
  - If **S** sends to **R** with latency **k**
  - Then **R** receives message when producing item that **S** sees in **k** of its own time steps



Sending Messages Upstream

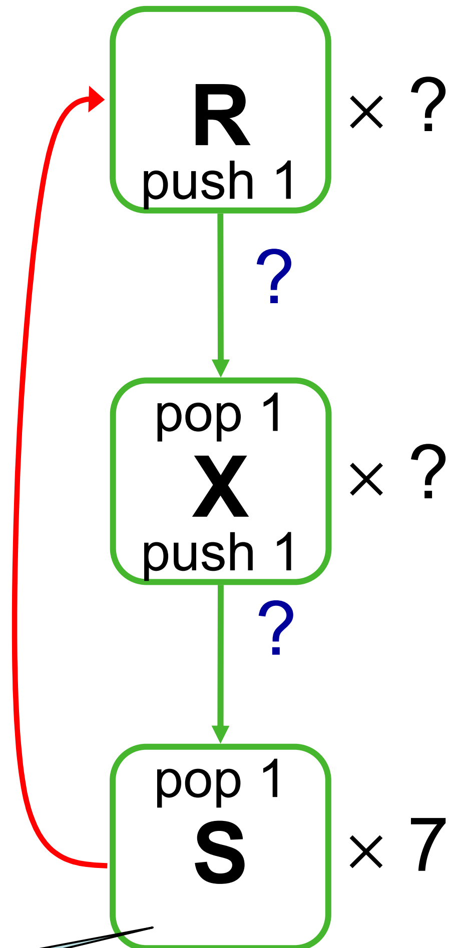
- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides provides a unified abstraction
- Intuition:
  - If **S** sends to **R** with latency **k**
  - Then **R** receives message when producing item that **S** sees in **k** of its own time steps



Receiver r;
r.decimate() @ [3:3]

Sending Messages Upstream

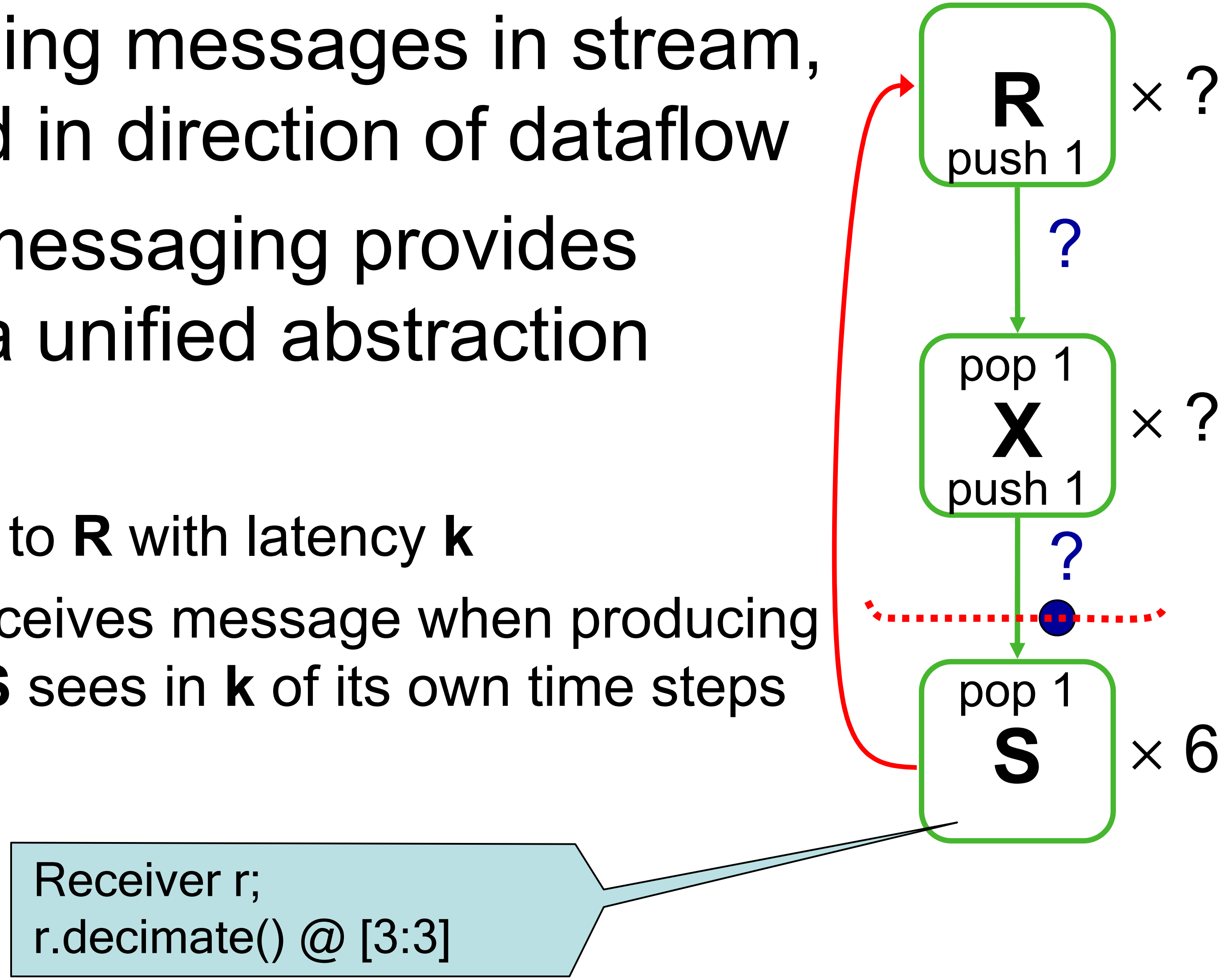
- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides provides a unified abstraction
- Intuition:
  - If **S** sends to **R** with latency **k**
  - Then **R** receives message when producing item that **S** sees in **k** of its own time steps



Receiver r;
r.decimate() @ [3:3]

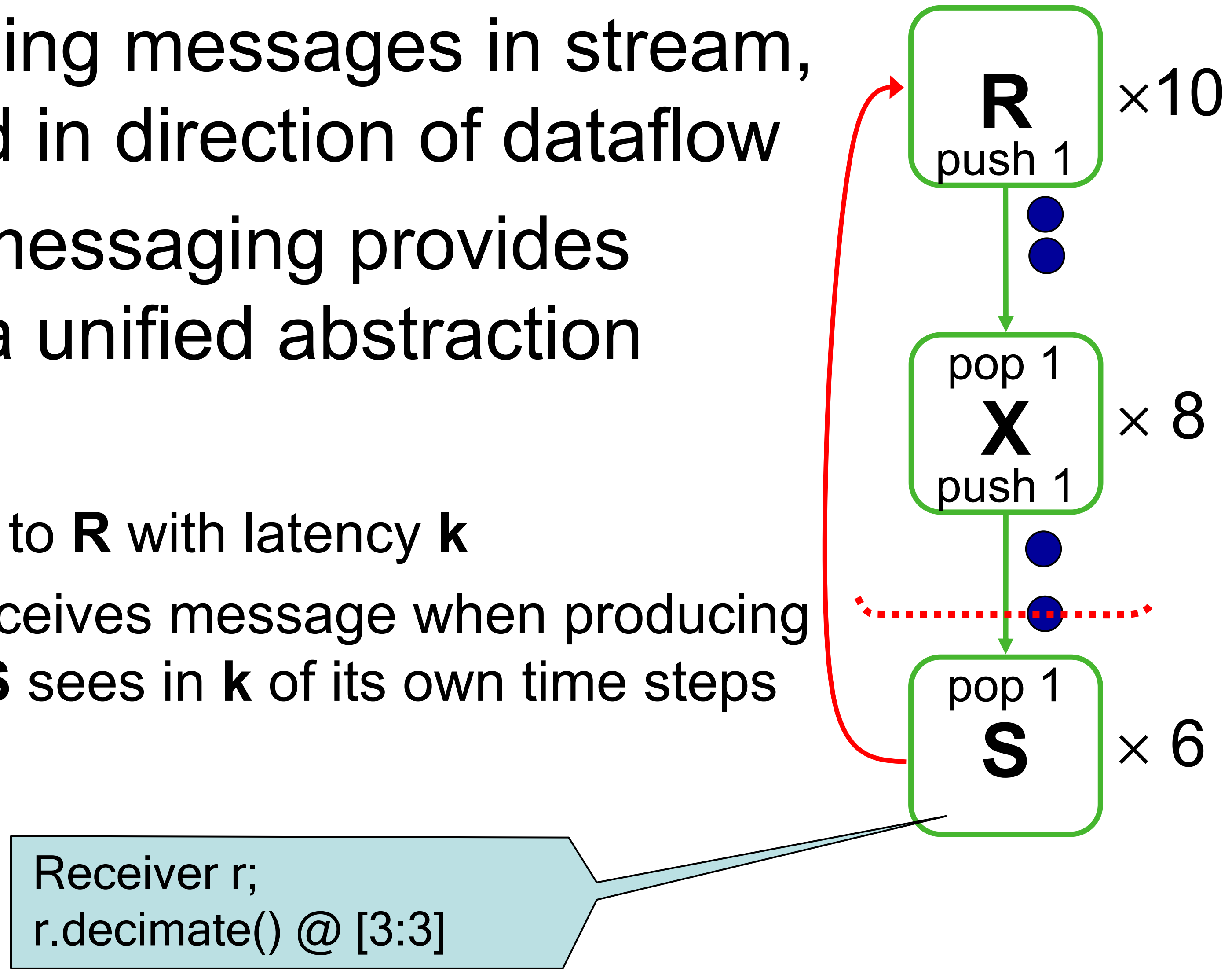
Sending Messages Upstream

- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides provides a unified abstraction
- Intuition:
  - If **S** sends to **R** with latency **k**
  - Then **R** receives message when producing item that **S** sees in **k** of its own time steps



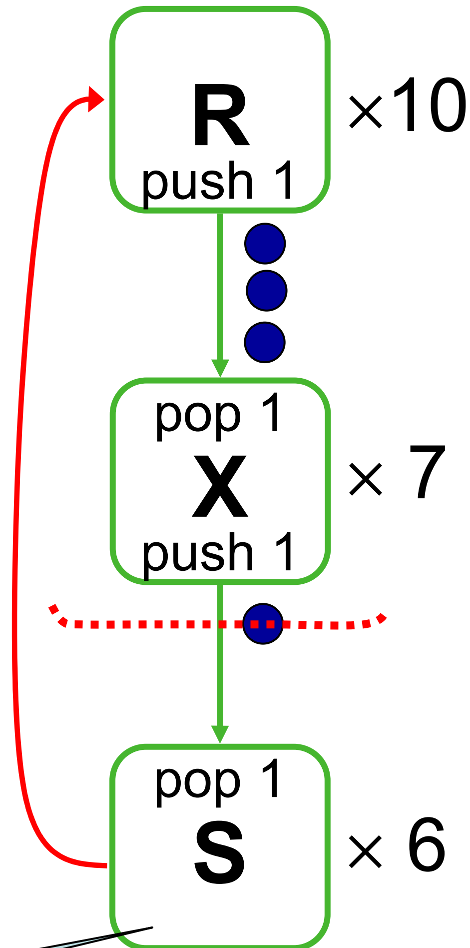
Sending Messages Upstream

- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides provides a unified abstraction
- Intuition:
  - If **S** sends to **R** with latency **k**
  - Then **R** receives message when producing item that **S** sees in **k** of its own time steps



Sending Messages Upstream

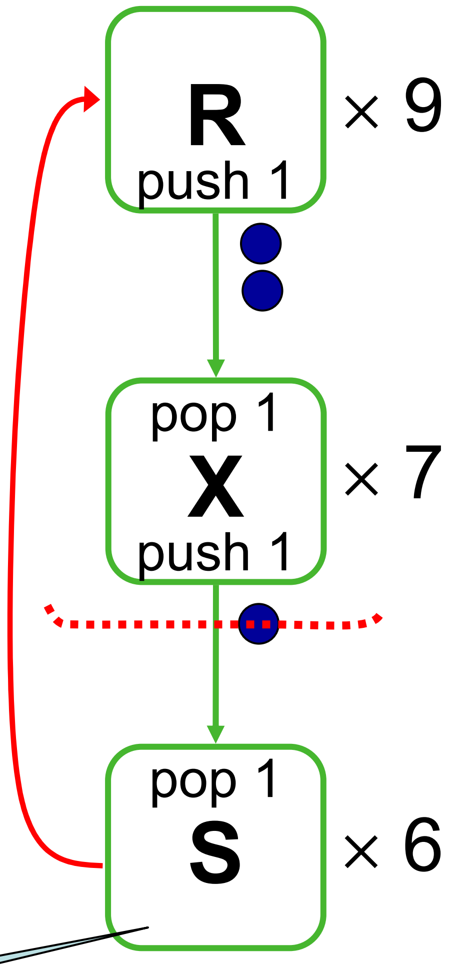
- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides provides a unified abstraction
- Intuition:
  - If **S** sends to **R** with latency **k**
  - Then **R** receives message when producing item that **S** sees in **k** of its own time steps



Receiver r;
r.decimate() @ [3:3]

Sending Messages Upstream

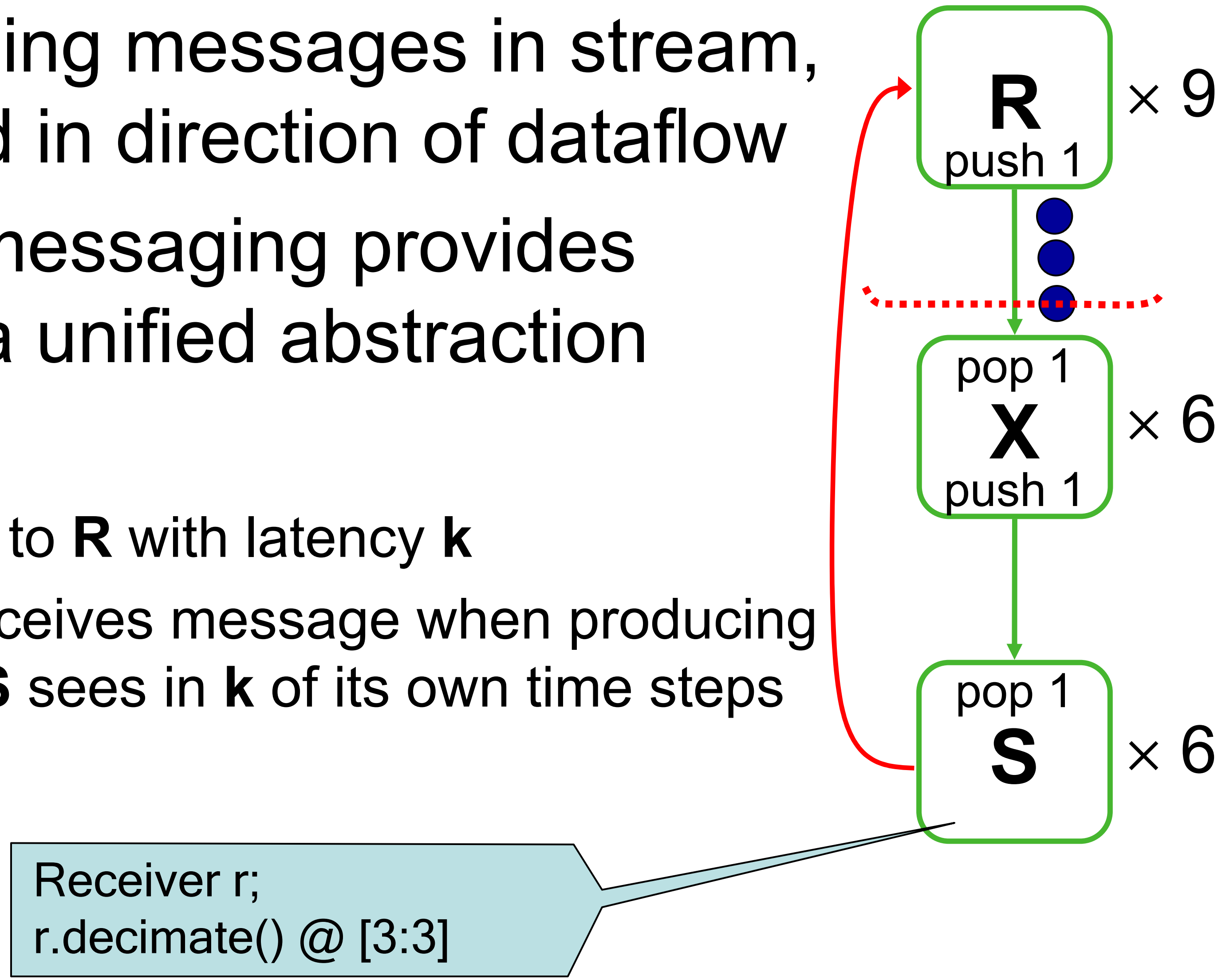
- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides provides a unified abstraction
- Intuition:
  - If **S** sends to **R** with latency **k**
  - Then **R** receives message when producing item that **S** sees in **k** of its own time steps



Receiver r;
r.decimate() @ [3:3]

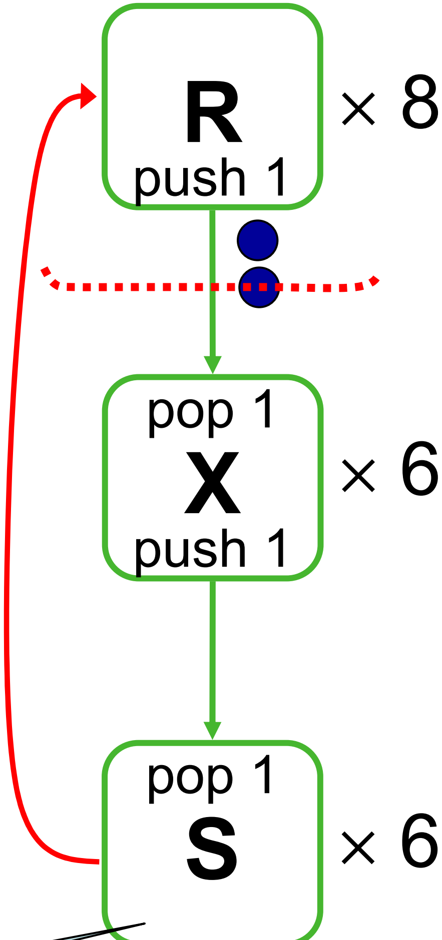
Sending Messages Upstream

- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides provides a unified abstraction
- Intuition:
  - If **S** sends to **R** with latency **k**
  - Then **R** receives message when producing item that **S** sees in **k** of its own time steps



Sending Messages Upstream

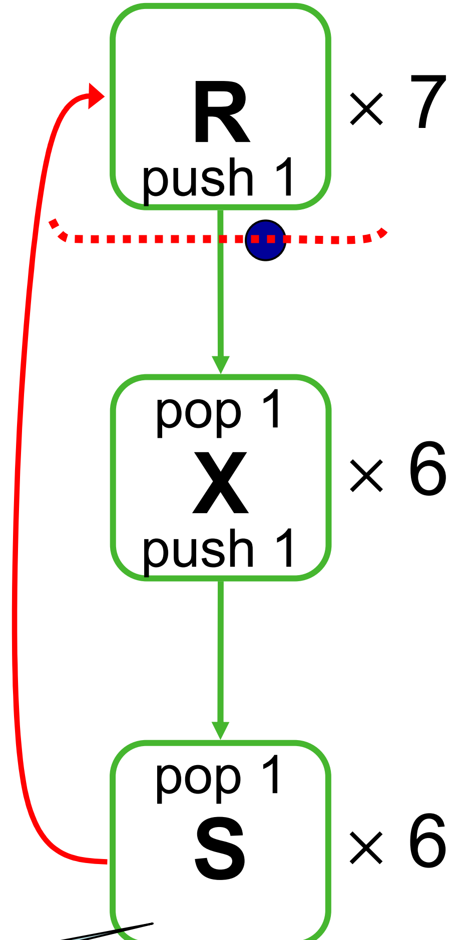
- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides provides a unified abstraction
- Intuition:
  - If **S** sends to **R** with latency **k**
  - Then **R** receives message when producing item that **S** sees in **k** of its own time steps




```
Receiver r;
r.decimate() @ [3:3]
```

Sending Messages Upstream

- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides provides a unified abstraction
- Intuition:
  - If **S** sends to **R** with latency **k**
  - Then **R** receives message when producing item that **S** sees in **k** of its own time steps

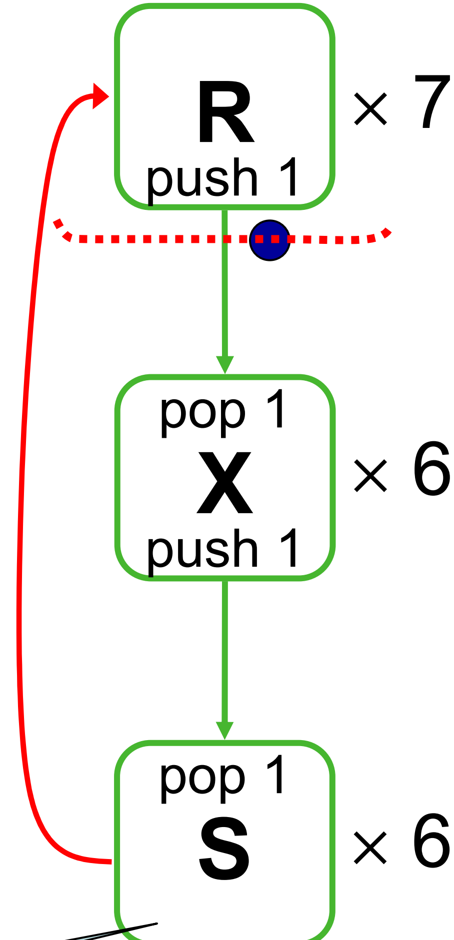


Receiver r;
r.decimate() @ [3:3]

Sending Messages Upstream

- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides provides a unified abstraction
- Intuition:
  - If **S** sends to **R** with latency **k**
  - Then **R** receives message when producing item that **S** sees in **k** of its own time steps

➡ **R** receives message on iteration 7



Receiver r;
r.decimate() @ [3:3]

Constraints Imposed on Schedule

	latency < 0	latency = 0	latency > 0
Message travels upstream			
Message travels downstream			

Constraints Imposed on Schedule

	latency < 0	latency = 0	latency > 0
Message travels upstream			Must not buffer too much data
Message travels downstream			

Constraints Imposed on Schedule

	latency < 0	latency = 0	latency > 0
Message travels upstream	Illegal	Illegal	Must not buffer too much data
Message travels downstream			

Constraints Imposed on Schedule

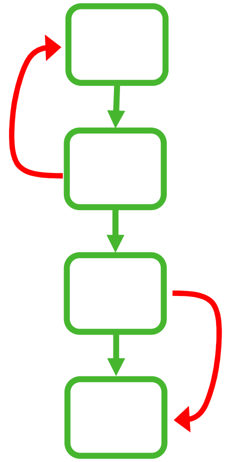
	latency < 0	**latency = 0**	**latency > 0**
Message travels upstream	Illegal	Illegal	Must not buffer too much data
Message travels downstream	Must not buffer too little data		

Constraints Imposed on Schedule

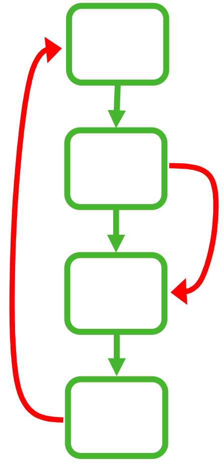
	latency < 0	**latency $= 0$**	**latency > 0**
Message travels upstream	Illegal	Illegal	Must not buffer too much data
Message travels downstream	Must not buffer too little data	No constraint	No constraint

Finding a Schedule

- Non-overlapping messages: greedy scheduling algorithm



- Overlapping messages: future work
  - Overlapping constraints can be feasible in isolation, but infeasible in combination

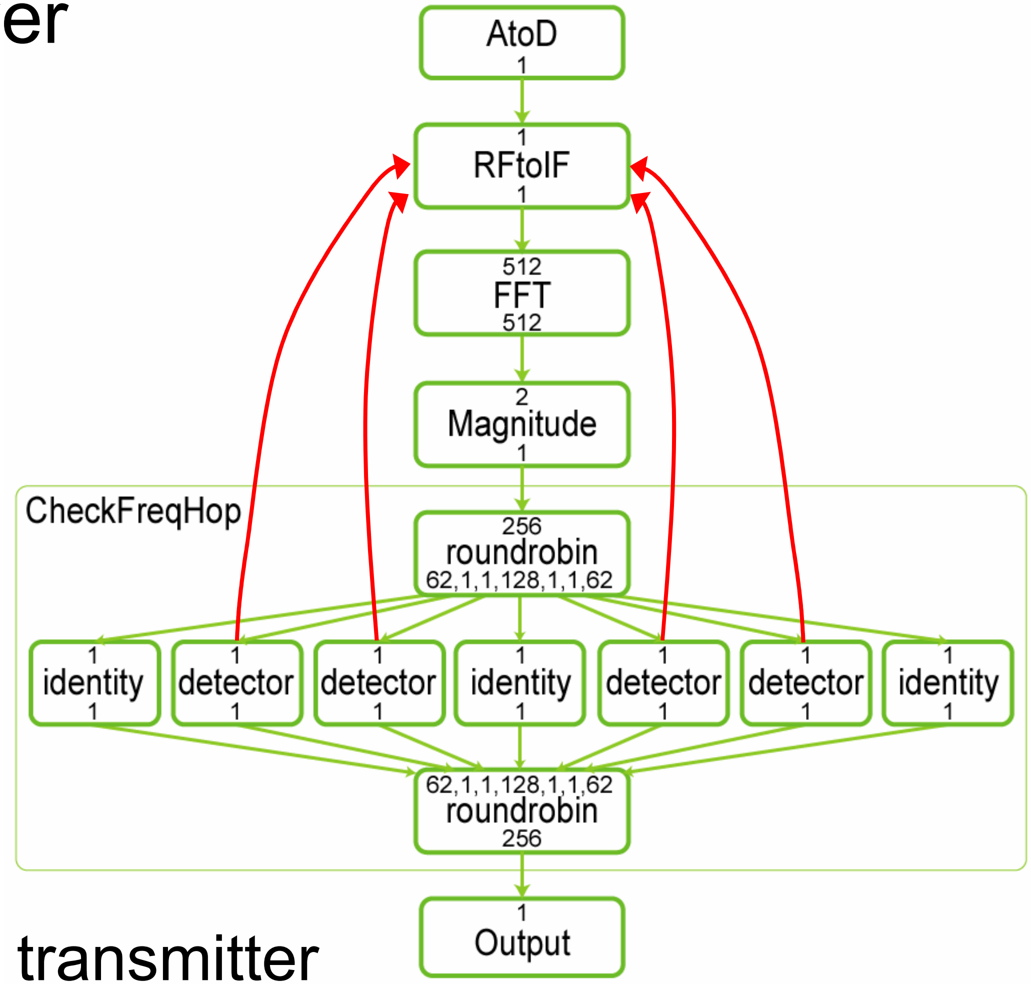


Outline

- StreamIt
- Teleport Messaging
- Case Study
- Related Work and Conclusion

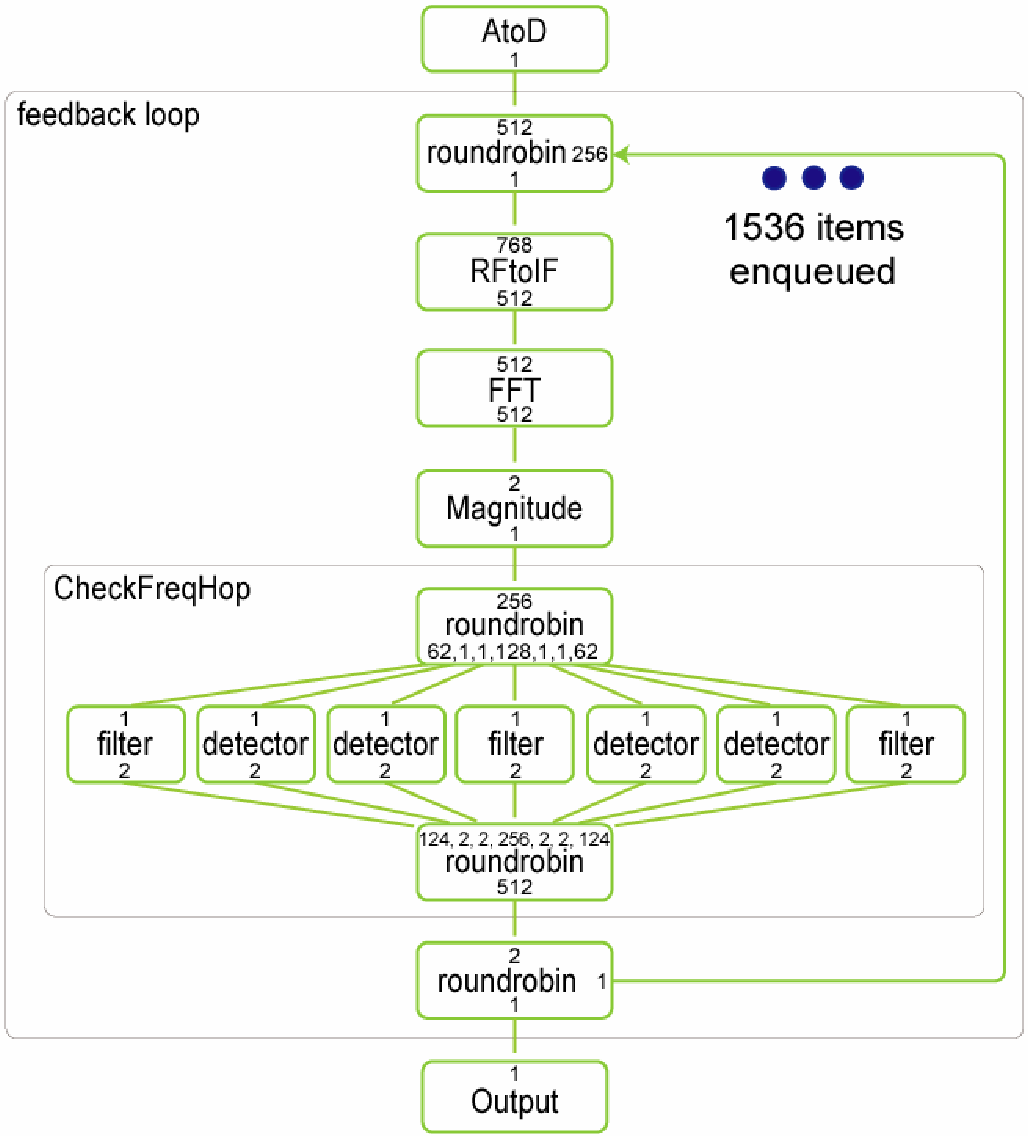
Frequency Hopping Radio

- Transmitter and receiver switch between set of known frequencies
- Transmitter indicates timing and target of hop using freq. pulse
- Receiver detects pulse downstream, adjusts RFtoIF with exact timing:
  - Switch at same time as transmitter
  - Switch at FFT frame boundary



Frequency Hopping Radio: Manual Feedback

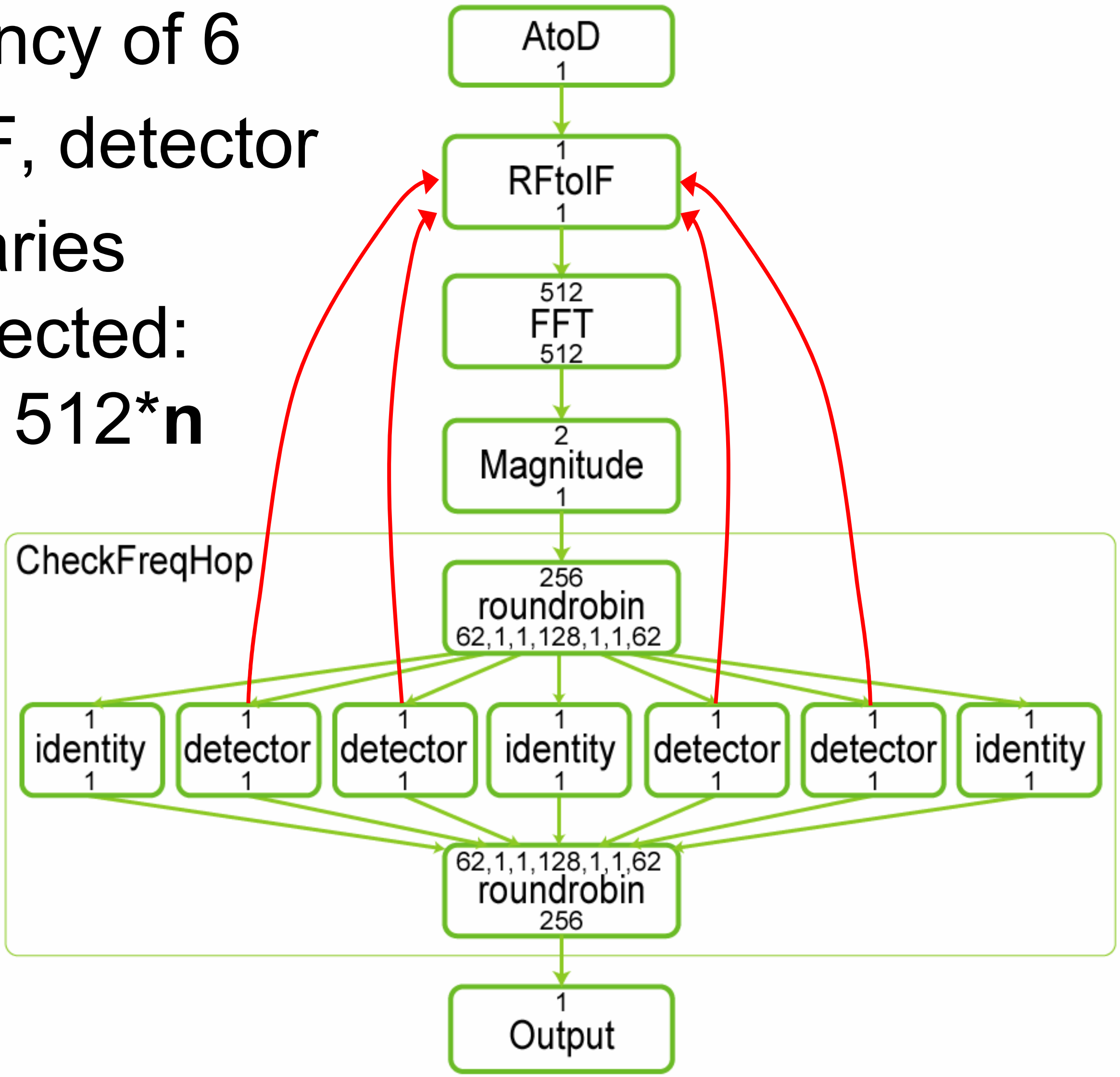
- Introduce feedback loop with dummy items to indicate presence or absence of message
- To add latency, enqueue 1536 initial items on loop
- Extra changes needed along path of message
  - Interleave messages, data
  - Route messages to loop
  - Adjust I/O rates
- To respect FFT frames, change RFtoIF granularity



Frequency Hopping Radio: Teleport Messaging

- Use message latency of 6
- Modify only RFtoIF, detector
- FFT frame boundaries automatically respected: $\mathrm{SDEP}_{\mathbf{RFIF}\leftarrow\mathbf{det}}(\mathbf{n}) = 512*\mathbf{n}$

➡ Teleport messaging improves programmability



Preliminary Results

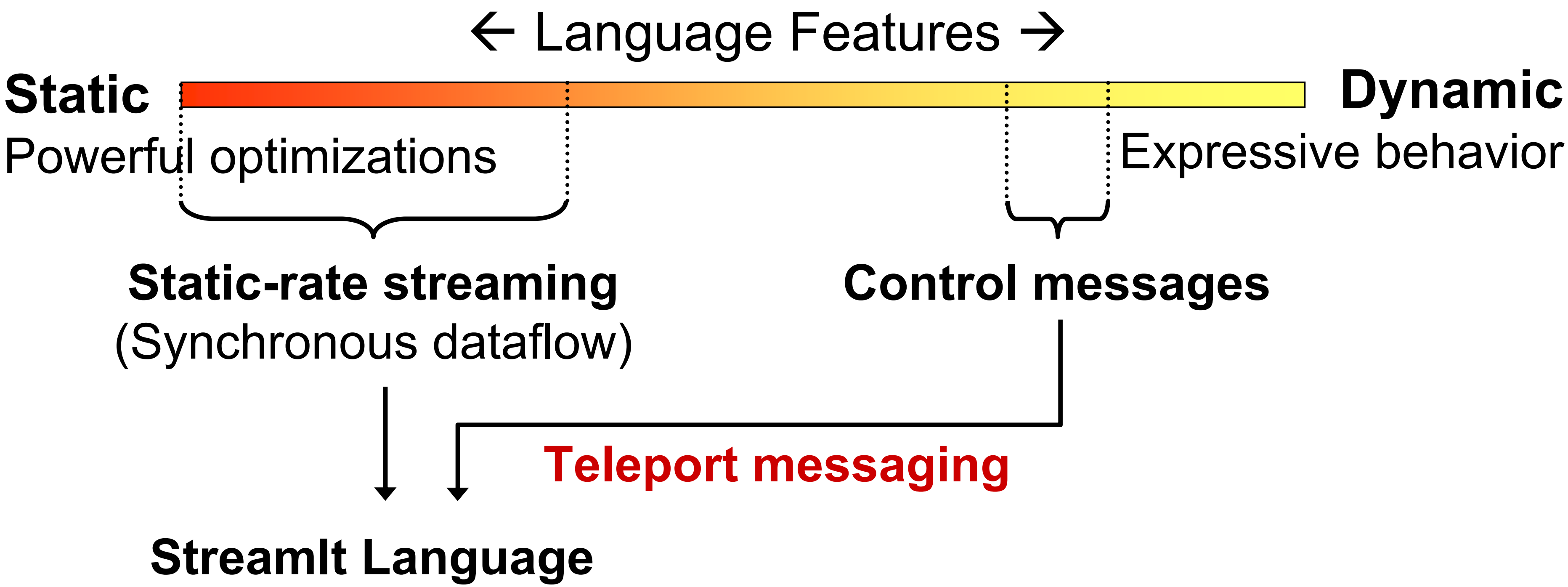
Outline

- StreamIt
- Teleport Messaging
- Case Study
- Related Work and Conclusion

Related Work

- Heterogeneous systems modeling
  - Ptolemy project (Lee et al.); scheduling (Bhattacharyya, …)
  - Boolean dataflow: parameterized data rates
  - Teleport messaging allows complete static scheduling
- Program slicing
  - Many researchers; see Tip'95 for survey
  - Like SDEP, find set of dependent operations
  - SDEP is more specialized; can calculate exactly
- Streaming languages
  - Brook, Cg, StreamC/KernelC, Spidle, Occam, Sisal, Parallel Haskell, Lustre, Esterel, Lucid Synchrone
  - Our goal: adding restricted dynamism to static language

Conclusion



- Teleport messaging provides precise and flexible event handling while allowing static optimizations
  - Data dependences (SDEP) is natural timing mechanism
  - Messaging exposes true communication to compiler

Extra Slides

Calculating SDEP in Practice

- Direct SDEP formulation:

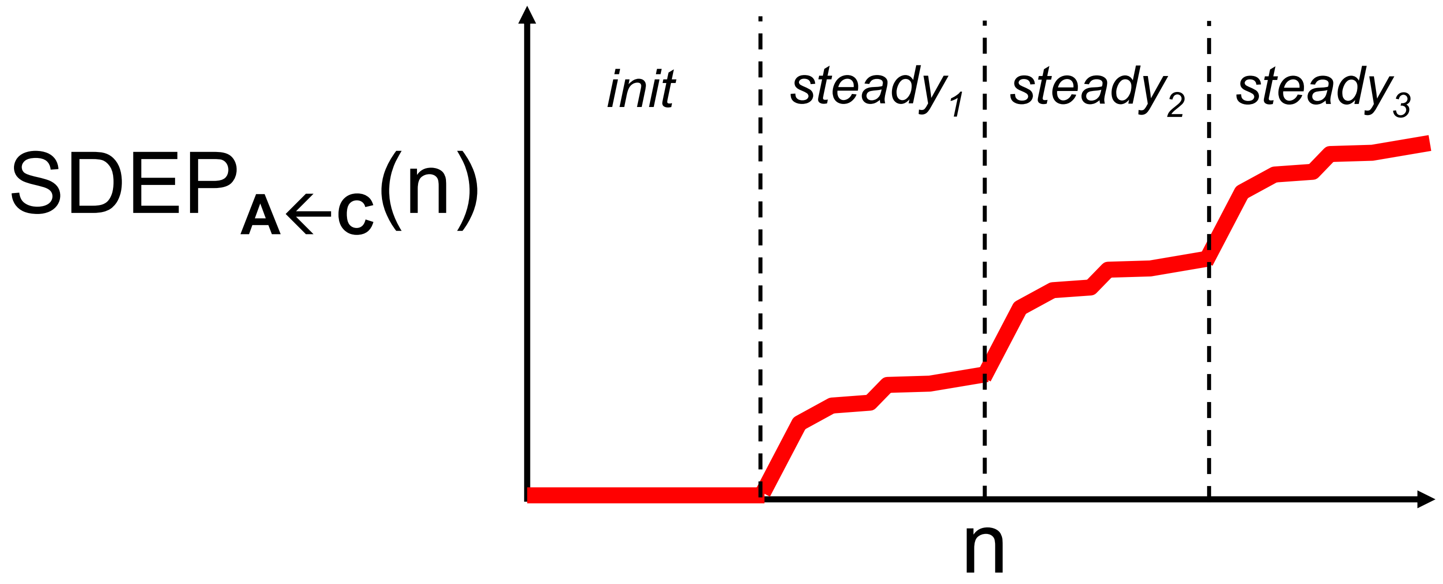
$$\mathrm{SDEP}_{A\leftarrow C}(n) = \max\left[\max\left(0, \left\lceil \frac{\max\left(0, \left\lceil \frac{n^{*}o_c - k}{u_{b1}} \right\rceil\right)^{*} o_{b1} - k}{u_a} \right\rceil\right),\right.$$

$$\max\left(0, \left\lceil \frac{\max\left(0, \left\lceil \frac{n^{*}o_c - k}{u_{b2}} \right\rceil\right)^{*} o_{b2} - k}{u_a} \right\rceil\right),$$

$$\left.\max\left(0, \left\lceil \frac{\max\left(0, \left\lceil \frac{n^{*}o_c - k}{u_{b3}} \right\rceil\right)^{*} o_{b3} - k}{u_a} \right\rceil\right)\right]$$

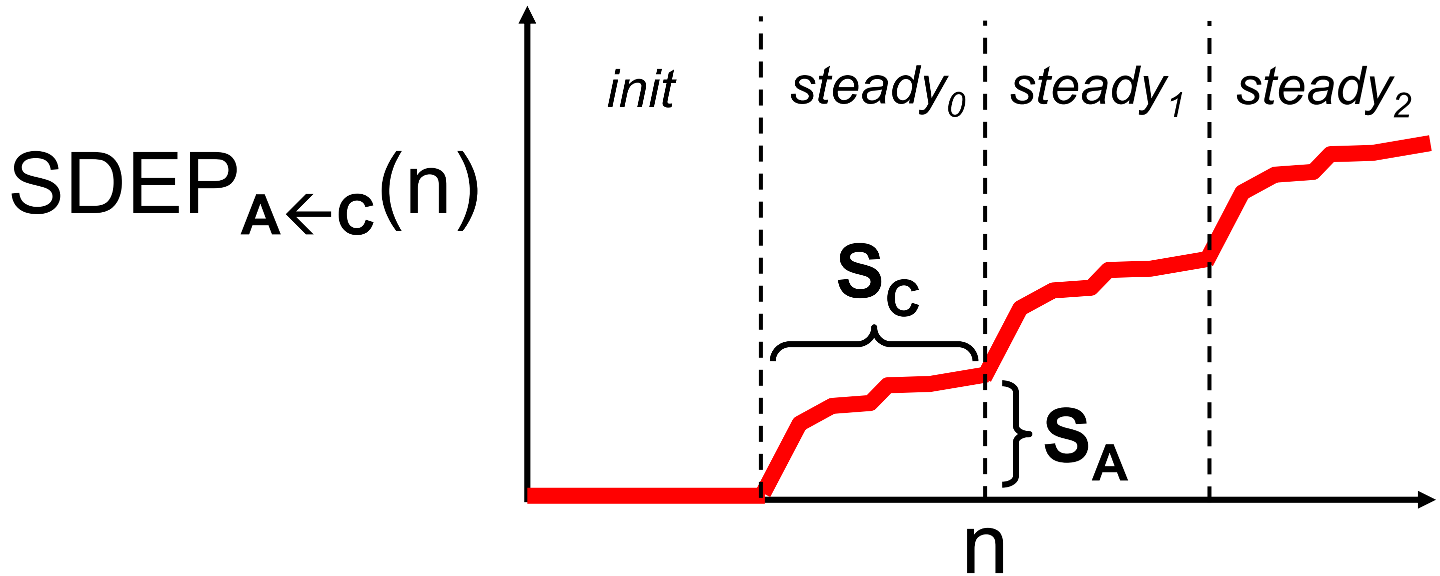
➡ Direct calculation could grow unwieldy

Calculating SDEP in Practice



- **initialization:** consumes all initial items
- **steady state:** repetition of each actor that does not change number of items on channels

Calculating SDEP in Practice



$$SDEP(n) = \begin{cases} 0 & n \in init \\ lookup\_table[n] & n \in steady_0 \\ k*S_A + SDEP(n - k*S_C) & n \in steady_k \end{cases}$$

➡ Build small SDEP table statically, use for all n

Sending Messages Upstream

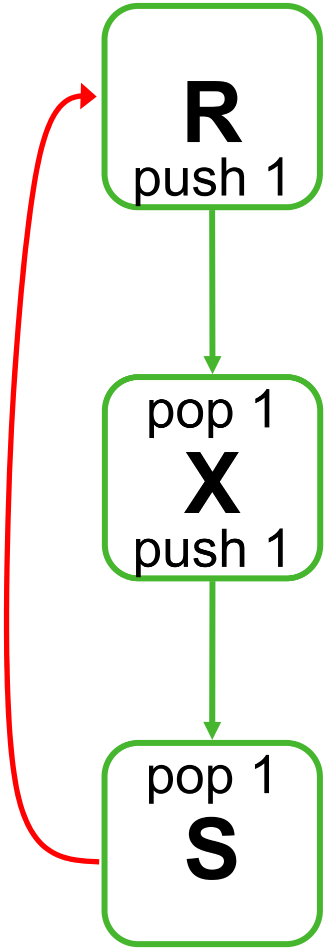
If **S** sends **upstream** message to **R**:

- with latency range $[\mathbf{k_1}, \mathbf{k_2}]$
- on the **n**th execution of **S**

Then message is delivered to **R**:

- on any iteration **m** such that

$$\mathrm{SDEP}_{\mathbf{R}\leftarrow\mathbf{S}}(\mathbf{n}+\mathbf{k_1}) \leq \mathbf{m} \leq \mathrm{SDEP}_{\mathbf{R}\leftarrow\mathbf{S}}(\mathbf{n}+\mathbf{k_2})$$



Sending Messages Upstream

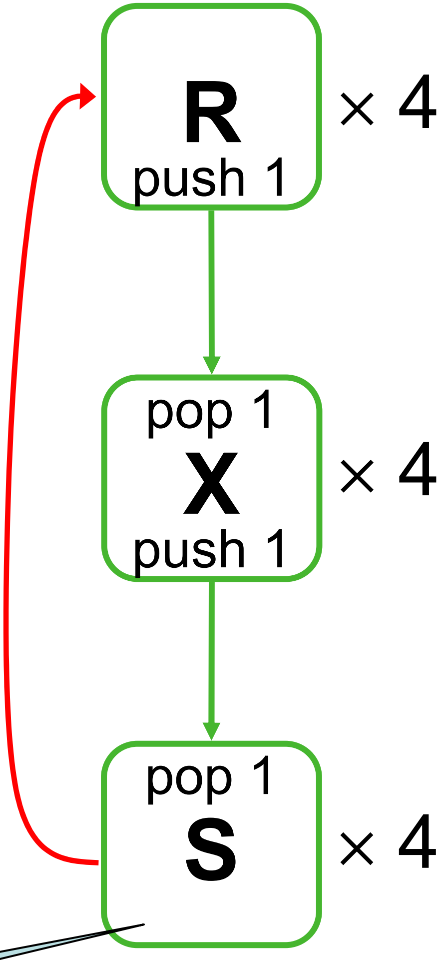
If **S** sends **upstream** message to **R**:

- with latency range [$\mathbf{k_1}$, $\mathbf{k_2}$]
- on the **n**th execution of **S**

Then message is delivered to **R**:

- on any iteration **m** such that

$\mathrm{SDEP}_{\mathbf{R \leftarrow S}}(\mathbf{n+k_1}) \leq \mathbf{m} \leq \mathrm{SDEP}_{\mathbf{R \leftarrow S}}(\mathbf{n+k_2})$



Receiver r;
r.decimate() @ [3:3]

Sending Messages Upstream

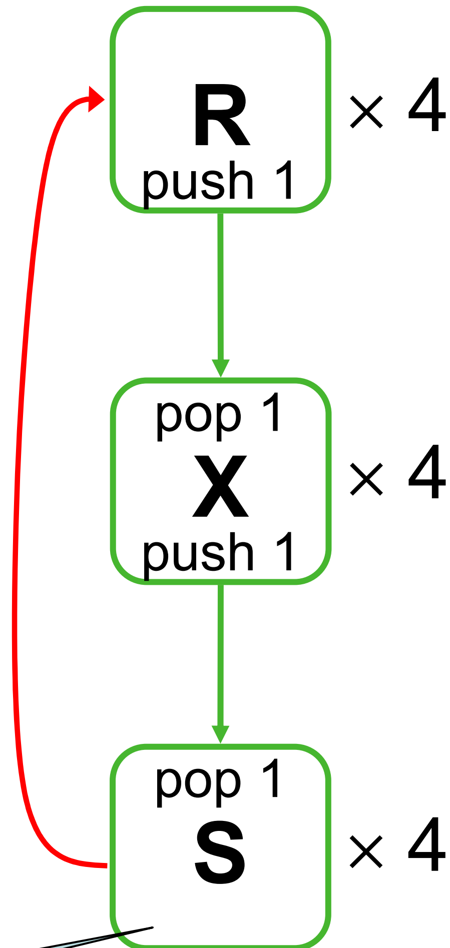
If **S** sends **upstream** message to **R**:

- with latency range **[3, 3]**
- on the **n**th execution of **S**

Then message is delivered to **R**:

- on any iteration **m** such that

$\mathrm{SDEP}_{\mathbf{R}\leftarrow\mathbf{S}}(\mathbf{n}+\mathbf{k_1}) \leq \mathbf{m} \leq \mathrm{SDEP}_{\mathbf{R}\leftarrow\mathbf{S}}(\mathbf{n}+\mathbf{k_2})$



Receiver r;
r.decimate() @ [3:3]

Sending Messages Upstream

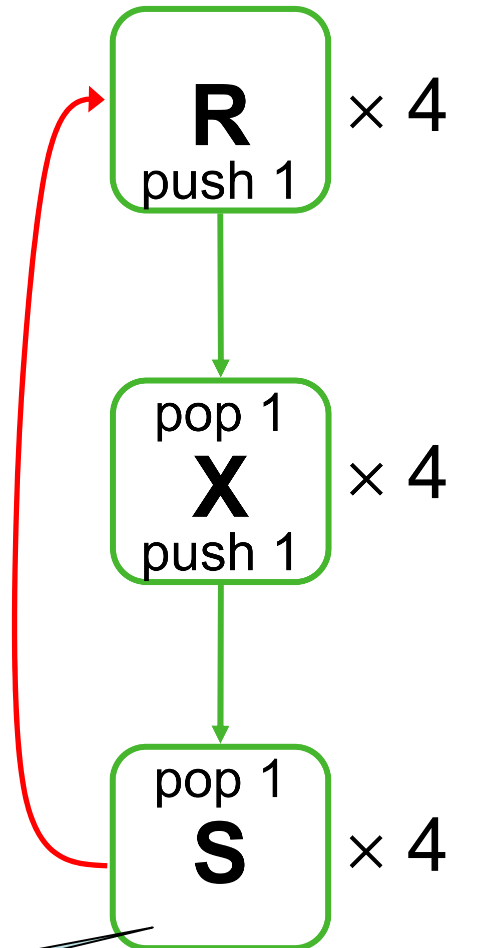
If **S** sends **upstream** message to **R**:

- with latency range [3, 3]
- on the 4th execution of **S**

Then message is delivered to **R**:

- on any iteration **m** such that

$$\mathrm{SDEP}_{\mathbf{R \leftarrow S}}(\mathbf{n+k_1}) \le \mathbf{m} \le \mathrm{SDEP}_{\mathbf{R \leftarrow S}}(\mathbf{n+k_2})$$




```
Receiver r;
r.decimate() @ [3:3]
```

Sending Messages Upstream

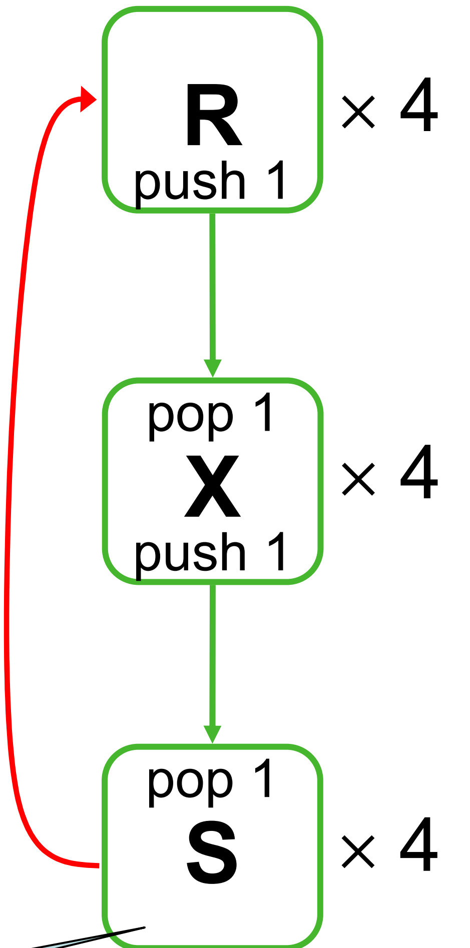
If **S** sends **upstream** message to **R**:

- with latency range **[3, 3]**
- on the **4**th execution of **S**

Then message is delivered to **R**:

- on any iteration **m** such that

$\mathrm{SDEP}_{\mathbf{R}\leftarrow\mathbf{S}}(\mathbf{4}+\mathbf{3}) \leq \mathbf{m} \leq \mathrm{SDEP}_{\mathbf{R}\leftarrow\mathbf{S}}(\mathbf{4}+\mathbf{3})$



Receiver r;
r.decimate() @ [3:3]

Sending Messages Upstream

If **S** sends **upstream** message to **R**:

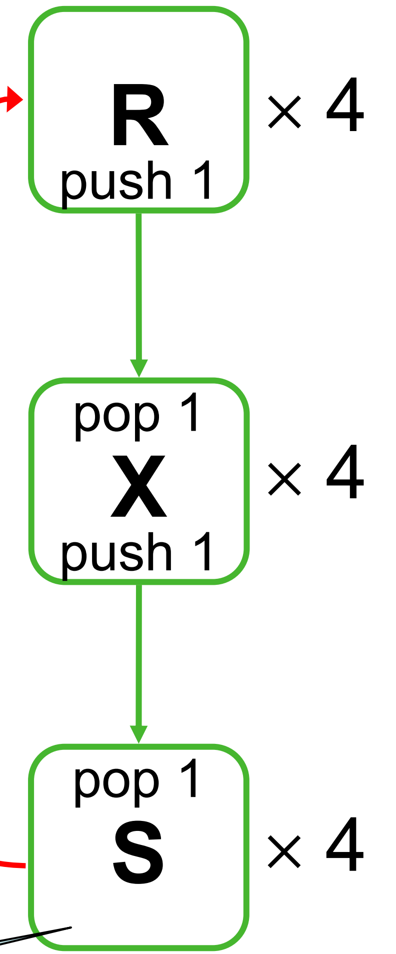
- with latency range [3, 3]
- on the 4th execution of **S**

Then message is delivered to **R**:

- on any iteration **m** such that

$$\mathrm{SDEP}_{R \leftarrow S}(4+3) \le \mathbf{m} \le \mathrm{SDEP}_{R \leftarrow S}(4+3)$$

$$\mathbf{m} = \mathrm{SDEP}_{R \leftarrow S}(7)$$




```
Receiver r;
r.decimate() @ [3:3]
```

Sending Messages Upstream

If **S** sends **upstream** message to **R**:

- with latency range **[3, 3]**
- on the **4**th execution of **S**

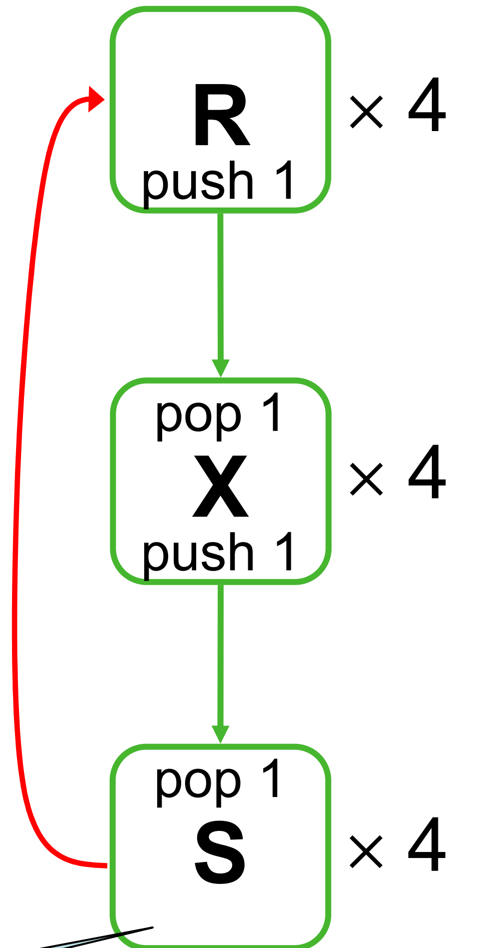
Then message is delivered to **R**:

- on any iteration **m** such that

$$\mathrm{SDEP}_{R\leftarrow S}(4+3) \leq \mathbf{m} \leq \mathrm{SDEP}_{R\leftarrow S}(4+3)$$

$$\mathbf{m} = \mathrm{SDEP}_{R\leftarrow S}(7)$$

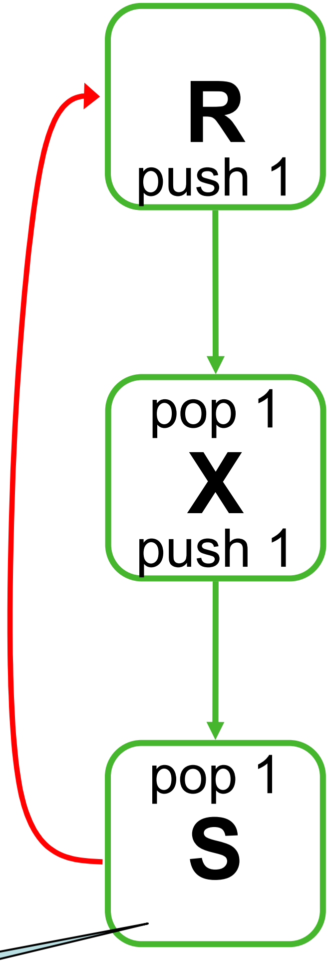
$$\mathbf{m} = 7$$



Receiver r;
r.decimate() @ [3:3]

Constraints Imposed on Schedule

- If **S** sends on iteration **n**, then **R** receives on iteration **n+3**
  - Thus, if **S** is on iteration **n**, then **R** must not execute past **n+3**
  - Otherwise, **R** could miss message
  - ➡ Messages constrain the schedule
- If latency is 0 instead of 3, then no schedule satisfies constraint
  - ➡ Some latencies are infeasible




```
Receiver r;
r.decimate() @ [3:3]
```

Implementation

- Teleport messaging implemented in cluster backend of StreamIt compiler
  - SDEP calculated at compile-time, stored in table
- Message delivery uses “credit system”
  - Sender sends two types of packets to receiver:
    1. **Credit:** “execute **n** times before checking again.”
    2. **Message:** “deliver this message at iteration **m**.”
  - Frequency of credits depends on SDEP, latency range
  - Credits expose parallelism, reduce communication

Evaluation

- Evaluation platform:
  - Cluster of 16 Pentium III's (750 Mhz)
  - Fully-switched 100 Mb network
- StreamIt cluster backend
  - Compile to set of parallel threads, expressed in C
  - Threads communicate via TCP/IP
  - Partitioning algorithm creates load-balanced threads