

Advanced Distributed Algorithms and Data Structures

Chapter 10: Communication Primitives

Christian Scheideler
Institut für Informatik
Universität Paderborn

Overview

- Broadcast
- Convergecast
- Anycast

Broadcast

Broadcast problem: send a message to all other processes in the system

- Process P_1 has a message M that it wants to send to processes P_2 to P_n .
- We assume that the processes form a clique.

Naive strategy:

- P_1 sends M directly to P_2 to P_n .

Problem: high runtime and communication work at P_1

Broadcast

Push Protocol:

- Every process that already got the message forwards it to a random process in each round.

Pseudo-code of Push protocol:

```
timeout: true →  
  if  $M \neq \perp$  then  
     $v := \text{random}(N)$   
     $v \leftarrow \text{push}(M)$ 
```

```
push(msg) →  
  if  $M = \perp$  then  
     $M := \text{msg}$ 
```

Initially, only P_1 holds the message and all other processes have $M = \perp$. As before, N contains connections to all other processes in the system.

Broadcast

Push Protocol:

- Every process that already got the message forwards it to a random process in each round.

Theorem 10.1: The Push Protocol needs $O(\log n)$ time and $O(n \log n)$ communication work, w.h.p., till all processes received the message.

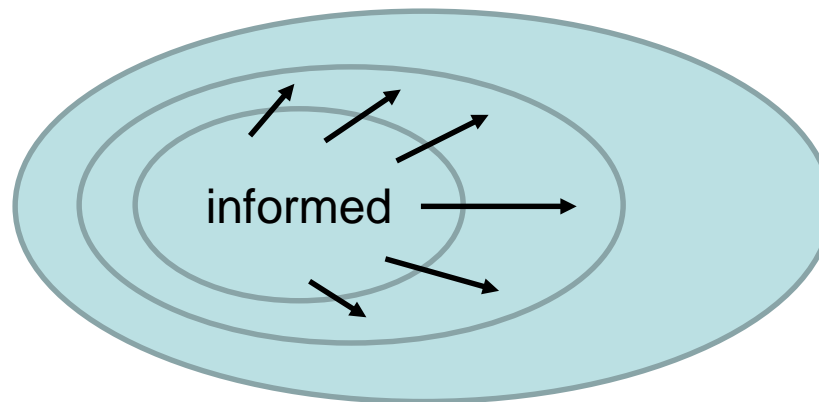
Proof:

- **Phase 1:** exponential progress till $n/2$ processes are informed.
- **Phase 2:** slow progress towards reaching all processes.

Broadcast

Analysis of Phase 1:

- **Phase 1a:** After $2c \cdot \log n$ rounds, P_1 has informed at least $c \cdot \log n$ other processes, w.h.p.
Proof: simple application of Chernoff bounds
- **Phase 1b:** As long as the number of informed processes is between $c \cdot \log n$ and $n/2$, the number of informed processes grows by a factor of at least $5/4$, w.h.p.



Broadcast

Analysis of Phase 1:

- **Phase 1a:** After $2c \cdot \log n$ rounds, P_1 has informed at least $c \cdot \log n$ other processes, w.h.p.
Proof: If this is not the case, a process must have been informed at least 3 times, which is very unlikely.
- **Phase 1b:** As long as the number of informed processes is between $c \cdot \log n$ and $n/2$, the number of informed processes grows by a factor of at least $5/4$, w.h.p.
Exercise: simple application of Chernoff bounds
- Phase 1b certainly takes just $O(\log n)$ rounds, w.h.p., till at least $n/2$ processes are informed, so altogether Phase 1 completes in $O(\log n)$ rounds, w.h.p.

Broadcast

Analysis of Phase 2:

- X_t : number of uninformed processes at the beginning of round t
- Suppose that $X_t \leq n/2$. Then it holds:
 $\Pr[\text{uninformed process not informed}] \leq (1-1/n)^{n/2} \leq e^{-1/2} \leq 0.61$
- Therefore, $E[X_{t+1}] \leq 0.61 \cdot X_t$
- As long as $X_t = \Omega(\log n)$, it follows from the Chernoff bounds that $X_{t+1} \leq 0.75 \cdot X_t$ w.h.p.
- Hence, after $O(\log n)$ rounds we are left with $O(\log n)$ uninformed processes w.h.p.
- Let P be one of these uninformed processes.
- $\Pr[P \text{ not informed in } c \cdot \ln n \text{ further rounds}] \approx ((1-1/n)^n)^{c \ln n} \leq (1/n)^c$
- Thus, after $O(\log n)$ rounds all processes are informed w.h.p.

Problem of phase 2: a lot of redundant message transmissions

Broadcast

- M : broadcast message
- A process is **notified** if it has received a NOTIFY message.
- A process is **informed** if it has received message M .

Initially, only the sink is notified and informed.

Push&Pull Protocol:

- Every notified process sends a NOTIFY(M) message to a random process in each round.
- The first time a process receives a NOTIFY(M) message, it sends an ACK(M) message back to the sender.
- If a process v got an ACK(M) message from a process w , it forwards M to w once it has received it.

Theorem 10.2: The Push&Pull Protocol needs $O(\log n)$ time and just $O(n)$ communication work for the broadcast message, with at most $O(\log n)$ broadcast message transmissions per process, till all processes received the message.

Proof:

- runtime and transmissions per process: follows from the proof of Theorem 10.1
- total communication work: every process will only receive the broadcast message once (since it only acknowledges the first NOTIFY message)

Broadcast

Pseudo-code of Push&Pull protocol: (P_1 : initially, notified=true and M set)

```
timeout: true →  
  if notified then  
    v:=random(N)  
    v←notify(in)
```

```
notify(out) →  
  if not notified then  
    out←ack(in)  
    delete out  
    notified:=true
```

```
ack(out) →  
  if  $M \neq \perp$  then  
    out←push(M)  
    delete out  
  else  
     $A := A \cup \{out\}$ 
```

```
push(msg) →  
  M:=msg  
  for all  $v \in A$  do  
    v←push(M)  
  delete v
```

Variables:

- notified: Boolean variable that is true if process has been notified
- A : stores all outgoing connections whose sinks have sent an ACK, so they need to be informed about M

Broadcast

Pseudo-code of Push&Pull protocol: (P_1 : initially, notified=true and M set)

```
timeout: true →  
  if notified then  
    v:=random(N)  
    v←notify(in)
```

```
notify(out) →  
  if not notified then  
    out←ack(in)  
    delete out  
    notified:=true
```

```
ack(out) →  
  if  $M \neq \perp$  then  
    out←push(M)  
    delete out  
  else  
     $A := A \cup \{out\}$ 
```

```
push(msg) →  
  M:=msg  
  for all  $v \in A$  do  
    v←push(M)  
  delete v
```

Remark: The code is not self-stabilizing (in a sense that the broadcast wouldn't work from an arbitrary state).

Exercise: think about a self-stabilizing version.

Broadcast

Advantage:

- Every process only receives the broadcast message once.
- Can also be adapted to asynchronous environments since it suffices for each notified process to contact $O(\log n)$ other processes before it stops sending further notifications.
- Protocol also works well for processes with different speeds (in a sense that the broadcast message is spread at the median timeout period of the processes, given that message transmissions are fast).

But:

- Communication work is not balanced among the processes since it may vary from $O(1)$ to $\Theta(\log n)$.
- Not robust to adversarial behavior (since adversarial processes may decide, for example, to drop the message).

Broadcast

Suppose that process P_0 initiates the broadcasting.

Solution to unbalanced communication work:

- P_0 cuts the broadcast message M into $k = \Omega(\log n)$ pieces M_1, \dots, M_k .
- For each i , P_0 sends M_i to a random process P_i .
- Processes P_1, \dots, P_k initiate the broadcasting of M_1, \dots, M_k in parallel using the Push&Pull protocol.
- It can be shown that in this case every process has a total communication work of $O(k)$ over all pieces, w.h.p., which is optimal.
- Also, when using error-correcting codes (e.g., Reed-Solomon codes), which only require the processes to receive a constant fraction of the k pieces to recover M , the solution above can tolerate a constant fraction of crash failures (i.e., the process simply stops working).

Broadcast

Suppose that process P_0 initiates the broadcasting.

Solution to unbalanced communication work:

- P_0 cuts the broadcast message M into $k = \Omega(\log n)$ pieces M_1, \dots, M_k .
- For each i , P_0 sends M_i to a random process P_i .
- Processes P_1, \dots, P_k initiate the broadcasting of M_1, \dots, M_k in parallel using the Push&Pull protocol.
- Another benefit of using error-correcting codes is that the slow phase (phase 2) in the Push&Pull protocol can be avoided:

Suppose that for a process to recover M it suffices to receive any $k/2$ out of k messages. Then it suffices to reach a point where every M_i has been sent to at least $3n/4$ processes so that every process can recover M w.h.p.

Why?

Broadcast

Alternative solution to adversarial behavior:

Careful Push&Pull Protocol:

- Every notified process sends a NOTIFY(M) message to a random process in each round.
- **Every time an uninformed process** receives a NOTIFY(M) message, it sends an ACK(M) message back to the sender.
- Once a process has been informed, it sends a NACK(M) message to all processes that have already notified it or will notify it in the future.
- If a process v got an ACK(M) message from a process w **and has not received a NACK(M) from it yet**, it forwards M to w once it has received it.

Remark: This protocol can handle a constant fraction of adversarial processes, and in contrast to the previous protocol (where the constant fraction depends on the choice of k and the redundancy of the error-correcting code), the constant fraction can be arbitrarily close to 1 if the protocol is run sufficiently long.

Overview

- Broadcast
- Convergecast
- Anycast

Convergecast

Convergecast problem: all processes want to send a message to some sink process.

Examples:

- The sink wants to know whether any one of the processes is currently in a specific state.
- The sink wants to know the current number of processes.
- The sink wants to know the sum of the values of the processes.
- The sink wants to determine the voting outcome of the processes.
- The sink wants to know whether a given predicate of the form $\bigvee_{v \in V} p(v)$ or $\bigwedge_{v \in V} p(v)$ is true, where $p(v)$ is a predicate depending only on the local state of process v .

Convergecast

Convergecast problem: all processes want to send a message to some sink process

Assumption: messages can be combined on their way to the sink process (see the examples).

Solution: sink process maintains a broadcast tree (using, for example, the tree built up by the ACK messages in the standard push&pull broadcast protocol), which will then be used by the other processes to send their values towards the sink.

Problem: adversarial processes may drop messages, may not report any value, or combine values in the wrong way

Convergecast

TCM model: adversary can only cause messages to be dropped. (If a TCL did not get a value from the AL, it may just use a default value.)

Simple adversarial model: a **fixed** set of processes (except the sink) is under DoS attack

Solution: just use the Push&Pull algorithm to build up a convergecast tree for the non-blocked processes, and then collect and combine the values along that tree.

Adaptive adversarial model: the adversary can block an arbitrary ε -fraction of the processes (except the sink) in each round

Convergecast

Consider the adaptive adversarial model.

Problem: The sink wants to know whether a given predicate p of the form $\bigvee_{v \in V} p(v)$ or $\bigwedge_{v \in V} p(v)$ is true, where $p(v)$ is a predicate depending only on the local state of process v .

Solution for $p = \bigvee_{v \in V} p(v)$: Convergecast problem reduces to broadcasting problem. We assume that all processes execute the Push protocol.

1. The sink initiates the computation of p via a broadcast request.
2. Once a process v has received the broadcast request, and $p(v) = \text{true}$, it initiates a broadcast of (p, true) . All will help spreading (p, true) messages via the Push protocol.
3. The sink waits for $O(\log n)$ many rounds. If it has never received a (p, true) message, it will output true , and otherwise it will output false .

Exercise: How to describe the protocol in pseudo-code.

Problem: If only a few processes v have $p(v) = \text{true}$, the adversary might be lucky with blocking these so that the sink sets p to false.

Convergecast

Consider the adaptive adversarial model.

Problem: The sink wants to know whether a given predicate p of the form $\bigvee_{v \in V} p(v)$ or $\bigwedge_{v \in V} p(v)$ is true, where $p(v)$ is a predicate depending only on the local state of process v .

Solution for $p = \bigvee_{v \in V} p(v)$: Convergecast problem reduces to broadcasting problem. We assume that all processes execute the Push protocol.

1. The sink initiates the computation of p via a broadcast request.
2. Once a process v has received the broadcast request, and $p(v) = \text{true}$, it initiates a broadcast of (p, true) . All will help spreading (p, true) messages via the Push protocol.
3. The sink waits for $O(\log n)$ many rounds. If it has never received a (p, true) message, it will output true , and otherwise it will output false .

Exercise: How to describe the protocol in pseudo-code.

Solution for $p = \bigwedge_{v \in V} p(v)$: Similar to above, but with true replaced by false and vice versa.

Convergecast

Problem: The sink wants to compute the sum S of the values stored in the processes.

Solution: The problem can be reduced to counting the number of processes and solving the following load balancing problem.

Load balancing problem: Given processes P_1, \dots, P_n with loads L_1, \dots, L_n , balance the loads so that in the end, every process has a load of $L = (1/n) \sum_{i=1}^n L_i$.

Once L is known, $S = \sum_{i=1}^n L_i$ can easily be computed because $S = n \cdot L$ and n is known because we assume the processes to form a clique.

Convergecast

Split&Combine Load Balancing:

- At the beginning of each round, each process v takes an ε -fraction of its current value and sends it to a randomly chosen process w . If w does not respond by the beginning of the next round, v adds this ε -fraction back to its value and sends a NACK message to w so that w deletes this ε -fraction. Otherwise, v sends an ACK message to w so that w adds this ε -fraction to its value.

Conjecture: After $O((1/\varepsilon)^2 \cdot \log n)$ many rounds, every process has a value of $(1 \pm O(\varepsilon))L$, w.h.p.

Remarks:

- The protocol should certainly work for a fixed set of blocked processes, but we conjecture that it also works for the adaptive adversarial model as long as the difference between the values is not too large (i.e., there are no extreme outliers that are blocked by the adversary by chance).
- Computing the exact value of L is difficult because there may always be values in transit.

Convergecast

Pseudo-code of split&combine protocol:

We need the following variables:

- **N**: as before neighborhood of a process
- **in**: incoming relay, as before
- **ack**: Boolean variable that is true if the reply of the contacted process v is received before the next timeout
- **val**: gives the value (or load) stored in the process
- **sentval**: gives the value to be transferred to the randomly selected process v
- **count**: assigns a unique number to each transfer attempt
- **S**: stores the set of transfer requests that are still active (i.e., no ACK or NACK has been received for these yet)

Convergecast

Pseudo-code of split&combine protocol:

```
timeout: true →  
  if ack=false then  
    val:=val+sentval  
    v←nack(in,count)  
  else  
    ack:=false  
  if val>0 then  
    count:=count+1  
    sentval:=ε·val  
    val:=val-sentval  
    v:=random(N)  
    v←push(in,count,sentval)
```

```
push(out,c,v) →  
  S:=S∪{(out,c,v)}  
  out←received(c)
```

```
received(c) →  
  if c=count then  
    ack:=true  
    v←ack(in,c)
```

```
ack(out,c) →  
  if ∃(out,c): (out,c,v)∈S then  
    val:=val+v  
    delete out; S:=S∖{(out,c,v)}
```

```
nack(out,c) →  
  if ∃(out,c): (out,c,v)∈S then  
    delete out; S:=S∖{(out,c,v)}
```

Convergecast

Requires FIFO delivery of messages (push received before nack!).

```
timeout: true →  
  if ack=false then  
    val:=val+sentval  
    v←nack(in,count)  
  else  
    ack:=false  
  if val>0 then  
    count:=count+1  
    sentval:=ε·val  
    val:=val-sentval  
    v:=random(N)  
    v←push(in,count,sentval)
```

```
push(out,c,v) →  
  S:=S∪{(out,c,v)}  
  out←received(c)
```

```
received(c) →  
  if c=count then  
    ack:=true  
    v←ack(in,c)
```

```
ack(out,c) →  
  if ∃(out,c): (out,c,v)∈S then  
    val:=val+v  
    delete out; S:=S∖{(out,c,v)}
```

```
nack(out,c) →  
  if ∃(out,c): (out,c,v)∈S then  
    delete out; S:=S∖{(out,c,v)}
```

Convergecast

Split&Combine Load Balancing:

- At the beginning of each round, each process v takes an ϵ -fraction of its current value and sends it to a randomly chosen process w . If w does not respond by the beginning of the next round, v adds this ϵ -fraction back to its value and sends a NACK message to w so that w deletes this ϵ -fraction. Otherwise, v sends an ACK message to w so that w adds this ϵ -fraction to its value.

Problem: If the rounds (i.e., timeouts) are executed too quickly, then the values are never successfully transferred. So a MIMD approach, for example, is needed to adjust the timeout periods so that an acknowledgement of a pushed value is received before the next timeout.

An interesting, alternative approach can be found here, for example:

Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. Proc. of the 6th Intl. Conference on Mobile Computing and Networking (MobiCom), pp. 56-67, 2000.

Overview

- Broadcast
- Convergecast
- Anycast

Anycast

Anycast problem: given a predicate p and a message M , send M to any process v with $p(v)=\text{true}$.

Examples:

- Send task to any idle process
- Send task to any process that is authorized or has the resources to execute it

Applications:

- Service discovery and auto-configuration (more flexible and robust than DHCP)
- Standardized by IETF (RFC 1546)

Anycast

Problem: send task to any idle process

Solution: work stealing

Basic idea: any process that is idle tries to steal a task from the pool of tasks

See also: Robert Blumofe and Charles Leiserson. Scheduling multithreaded computations by work stealing. In Proc. of the 35th Symp. on Foundations of Computer Science (FOCS), pp. 356-368, 1994.

Anycast

Problem: send task to any idle process

Push&Pull protocol:

- every busy process that has a task tries to push it to a random neighbor,
- every idle process regularly sends a pull request to a random neighbor to ask for a task, and
- any idle process receiving a task processes it and becomes busy.

Works fine if there are many tasks or many idle processes since chances are high in this case that a task is pushed to an idle process or that an idle process pulls a task.

Anycast

Problem: send task to any idle process

Push&Pull protocol:

- every busy process that has a task tries to push it to a random neighbor,
- every idle process regularly sends a pull request to a random neighbor to ask for a task, and
- any idle process receiving a task processes it and becomes busy.

Problem: If there are only a few idle processes and only a few tasks, then it may take quite some time until all tasks are processed resp. all idle processes have found a task.

Anycast

Problem: send task to any idle process, but there are few tasks and idle processes

Simple solution: suppose that we have a leader process v (which can be determined with the help of the median rule, for example).

1. v uses the notification mechanism in the Push&Pull broadcast protocol to set up a convergecast tree to v .
2. All tasks are sent towards v , and all idle processes send an idle token towards v in that tree.
3. Whenever a task meets an idle token on its way to v or at v , the task is sent to the origin of the idle token (which is easy to do if the token contains a relay to v).

Anycast

Problem: send task to any idle process, but there are few tasks and idle processes

Alternative solution: Convergecast-Pull protocol

- Every idle process sends a pull request to a random neighbor in each round.
- The pull requests will create convergecast trees (with the constraint that every busy as well as idle process participates in at most c of them) along which the tasks will be sent to the idle processes.
- Every idle process that receives a task acknowledges that to the sender, becomes busy and deletes its convergecast tree.
- Every busy process that is not yet part of a convergecast tree sends any tasks it has to random neighbors.

Problem: in this case several tasks might be directed to an idle process at the same time, which is not the case for the previous protocol.

Anycast

Pseudocode of convergecast-pull protocol:

```
timeout: true →
  for any  $u \in S$  do
    if  $u.\text{sink} = \text{task-in}$  or  $u.\text{sink} = \perp$  or
        $\exists v \in S \setminus \{u\}: v.\text{sink} = u.\text{sink}$  then
       $S := S \setminus \{u\}$ ; delete  $u$ 
    else
       $v := \text{random}(N)$ 
       $v \leftarrow \text{pull}(u)$ 
  if  $\text{Tasks} \neq \emptyset$  then
    if  $\text{idle}$  then
      remove any task from  $\text{Tasks}$ 
       $\text{idle} := \text{false}$ 
      delete  $\text{task-in}$  { deletes relay tree }
    else
       $t := \text{random}(\text{Tasks})$ ;  $\text{Tasks} := \text{Tasks} \setminus \{t\}$ 
      if  $S \neq \emptyset$  then
         $v := \text{random}(S)$ 
         $v \leftarrow \text{push}(t)$ 
         $S := S \setminus \{v\}$ ; delete  $v$  { deletes subtree of  $v$  }
      else
         $v := \text{random}(N)$ 
         $v \leftarrow \text{push}(t)$ 
  if  $\text{idle}$  then
     $v := \text{random}(N)$ 
     $v \leftarrow \text{pull}(\text{task-in})$ 
```

```
pull(out) →
  if  $|S| < c$  then
     $S := S \cup \{\text{out}\}$ 
  else
    delete  $\text{out}$ 
```

```
push(t) →
   $\text{Tasks} := \text{Tasks} \cup \{t\}$ 
```

Variables used in the protocol:

- task-in : sink of convergecast tree
- Tasks : set of tasks in process
- S : set of relays of convergecast trees

Remark: this protocol is not self-stabilizing.
Why?

Anycast

Pseudocode of convergecast-pull protocol:

```
timeout: true →
  for any  $u \in S$  do
    if  $u.\text{sink} = \text{task-in}$  or  $u.\text{sink} = \perp$  or
        $\exists v \in S \setminus \{u\}: v.\text{sink} = u.\text{sink}$  then
       $S := S \setminus \{u\}$ ; delete  $u$ 
    else
       $v := \text{random}(N)$ 
       $v \leftarrow \text{pull}(u)$ 
  if  $\text{Tasks} \neq \emptyset$  then
    if  $\text{idle}$  then
      remove any task from  $\text{Tasks}$ 
       $\text{idle} := \text{false}$ 
      delete  $\text{task-in}$  { deletes relay tree }
    else
       $t := \text{random}(\text{Tasks})$ ;  $\text{Tasks} := \text{Tasks} \setminus \{t\}$ 
      if  $S \neq \emptyset$  then
         $v := \text{random}(S)$ 
         $v \leftarrow \text{push}(t)$ 
         $S := S \setminus \{v\}$ ; delete  $v$  { deletes subtree of  $v$  }
      else
         $v := \text{random}(N)$ 
         $v \leftarrow \text{push}(t)$ 
  if  $\text{idle}$  then
     $v := \text{random}(N)$ 
     $v \leftarrow \text{pull}(\text{task-in})$ 
```

```
pull(out) →
  if  $|S| < c$  then
     $S := S \cup \{\text{out}\}$ 
  else
    delete  $\text{out}$ 
```

```
push(t) →
   $\text{Tasks} := \text{Tasks} \cup \{t\}$ 
```

**Problem: Tasks in transit may get deleted
if convergecast tree is deleted!**

Maybe, add another primitive that allows
a request to surface at the process where
it cannot be sent any further?

Anycast

General anycast problem: given a predicate p and a message M , send M to any process v with $p(v)=\text{true}$.

Solution: load balancing

See also: Baruch Awerbuch, André Brinkmann, and Christian Scheideler. Anycasting in Adversarial Systems: Routing and Admission Control. In Proc. of ICALP 2003, pp. 1153-1168.

Anycast

Basic assumptions:

- All messages are **atomic**
- Time proceeds in **synchronous** rounds
- Only **point-to-point** connections (links)
- Each link can forward **one** message per round.
- Information exchange between neighbors:
0 cost

Anycast

Adversarial anycasting model: In each round, the adversary can

- propose an arbitrary set of directed edges with at most Δ incoming and outgoing edges per node and
- inject any set of messages.

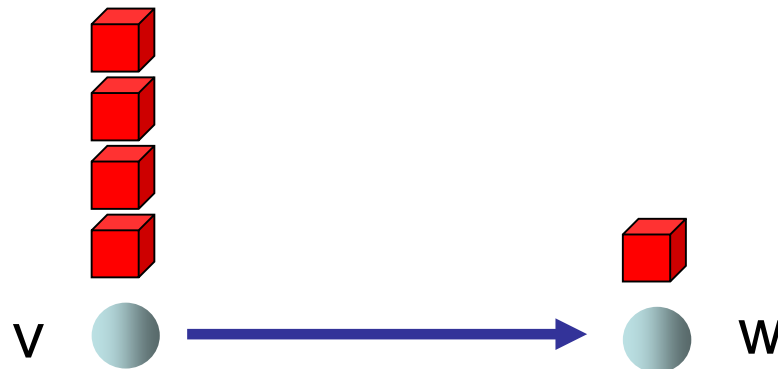
Goal: compare number of message deliveries (**throughput**) with optimal algorithm (**OPT**)

Algorithm is **(c,s)-competitive**: reaches **c**-fraction of optimal throughput with **s** times more buffer space per anycast address than **OPT**

Anycast

Basic approach: for every time step t and every proposed edge (v,w) ,

- $\#msgs(v) - \#msgs(w) > T$: send message
- receive all incoming and injected messages.

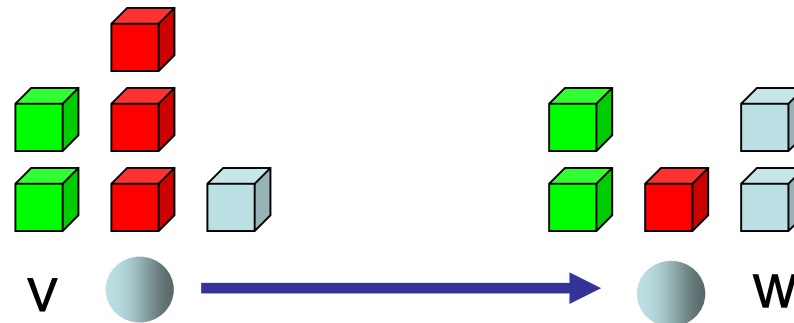


Anycast

Original model \rightarrow option set model:

Original model:

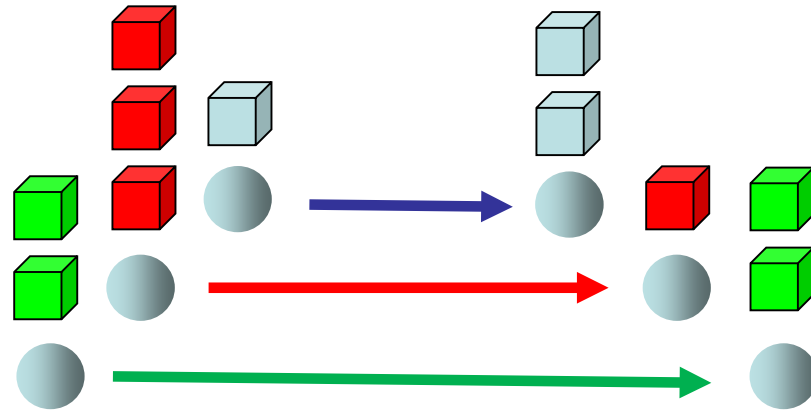
- Each node has a buffer for each anycast address
- Each round the adversary proposes edges



Anycast

Option set model:

- Buffer \rightarrow virtual node
- Edge \rightarrow set of virtual edges between corresponding buffer pairs, only one edge can be taken by algorithm



Anycast

Anycast approach in option set model:

Balancing algorithm:

For every time step t and every option set S ,

- select edge $(v,w) \in S$ with largest $\#msg(v) - \#msg(w)$
- $\#msg(v) - \#msg(w) > T$: send message along (v,w)
- receive all incoming and injected messages (if not possible because the buffer is full: delete any message), and absorb all messages at destination

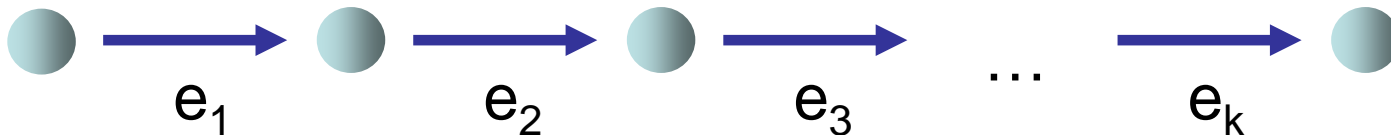
Theorem: The balancing algorithm is $(1-\epsilon, O(L/\epsilon))$ -competitive, where L is the average path length used by successful messages in OPT.

Anycast

Proof:




- **B**: buffer size used by OPT for every anycast address
- **L**: average path length used by OPT
- Each message that is successful in OPT has a schedule

$$S = (t_0, (t_1, e_1), (t_2, e_2), \dots, (t_k, e_k))$$



Anycast

Proof (continued):

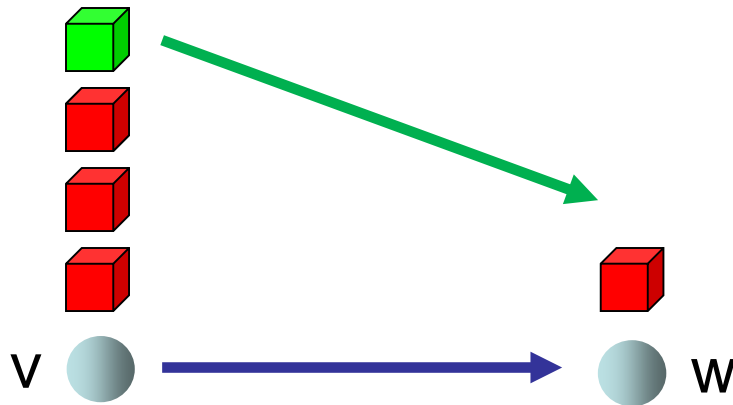
- OPT has buffer size B : at most B schedules have their current position at a node
- Balancing algorithm with buffer size $H > B$:
We distinguish between 3 types of messages:
 - **Representatives**: on schedule 
 - **Zombies**: have no schedule 
 - **Losers**: lost contact to schedule 

Anycast

Rules for messages:

- Representatives (green) always try to keep up with OPT message

Schedule edge offered:



Edge is used by
balancing algorithm:
representative keeps up

Anycast

Rules for messages:

- Representatives (green) always try to keep up with OPT message

Schedule edge offered:



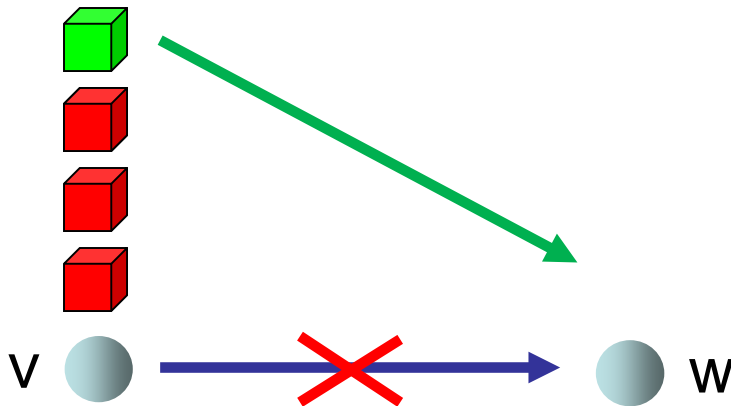
Edge is used by
balancing algorithm:
representative keeps up

Anycast

Rules for messages:

- Representatives (green) always try to keep up with OPT message

Schedule edge offered:



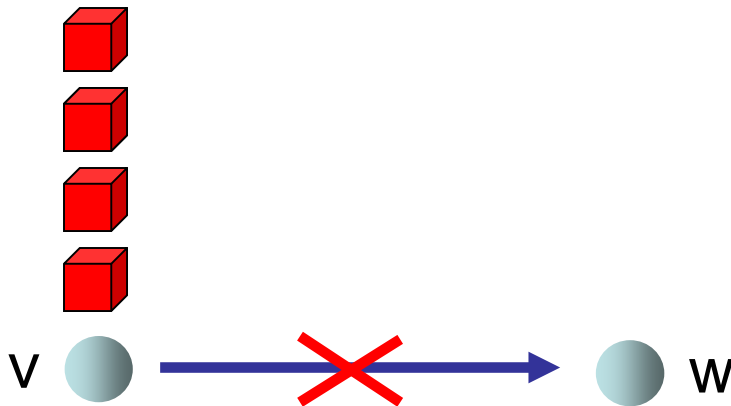
Edge not used, no loser available at w :
turn representative into loser

Anycast

Rules for messages:

- Representatives (green) always try to keep up with OPT message

Schedule edge offered:



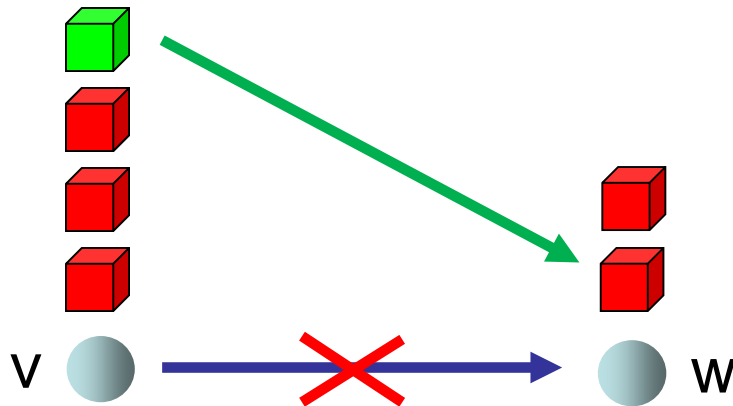
Edge not used, no loser available at w :
turn representative into loser

Anycast

Rules for messages:

- Representatives (green) always try to keep up with OPT message

Schedule edge offered:



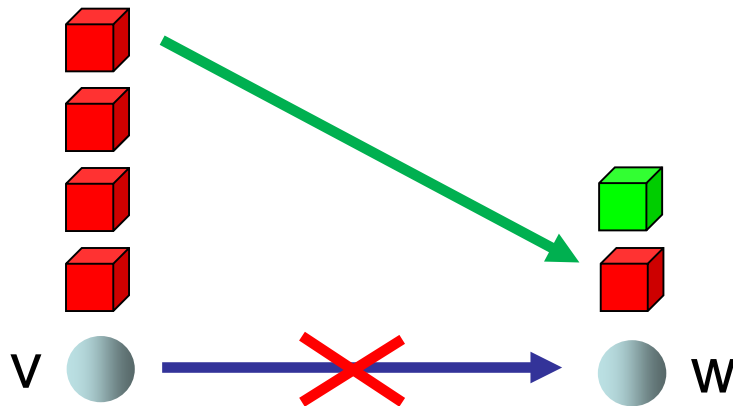
Edge not used, but loser available at **w**:
replacement of roles

Anycast

Rules for messages:

- Representatives (green) always try to keep up with OPT message

Schedule edge offered:



Edge not used, but loser available at **w**:
replacement of roles

Anycast

Rules for messages:

- Zombies only exist at heights $H-B+1, \dots, H$
- If zombie below $H-B+1$: converted into **loser**
- Messages stored in particular order



representatives



zombies



losers



Anycast

- h_v : **height** of node v (# losers)
- ϕ_v : **potential** of node v

$$\phi_v = \sum_{h=1}^{h_v} h$$

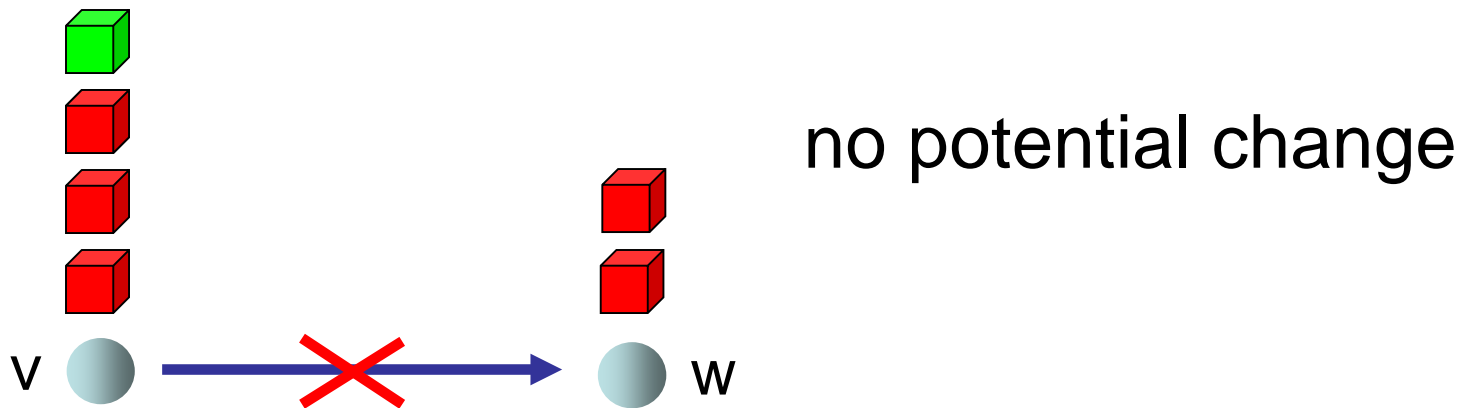
- $\Phi = \sum_v \phi_v$: potential of system

Goal: use potential to bound number of deleted packets compared to OPT

Anycast

If $T > B + 2\Delta - 1$:

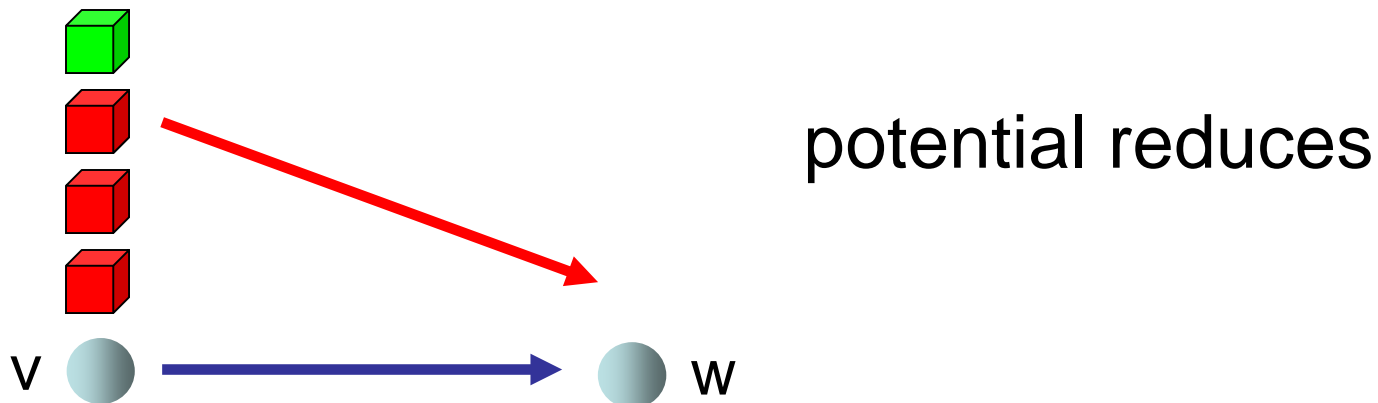
- Any option set not containing a schedule edge in OPT does **not increase** potential
- Case 1: no message moved by BA



Anycast

If $T > B + 2\Delta - 1$:

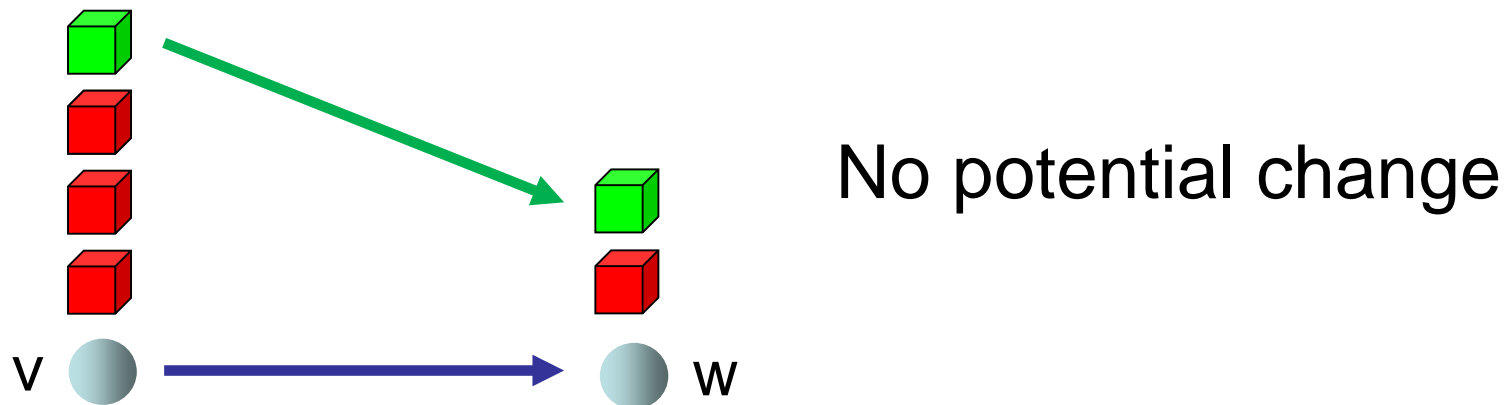
- Any option set not containing a schedule edge in OPT does **not increase** potential
- Case 2: message moved: move loser



Anycast

If $T > B + 2\Delta - 1$:

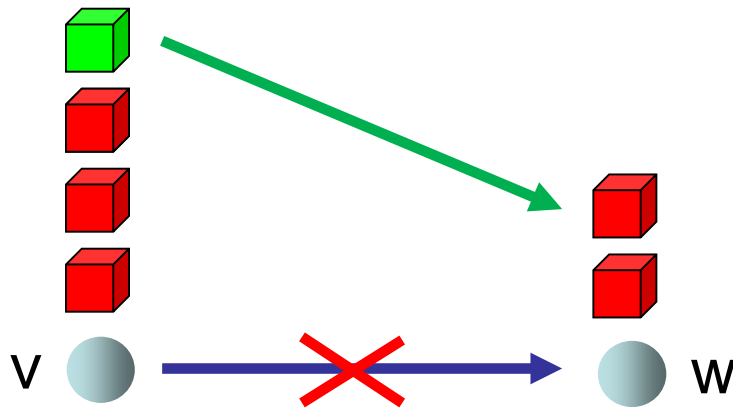
- Any option set containing a schedule edge **increases** potential by at most $2B + 3\Delta$
- Case 1: message moved: move rep.



Anycast

If $T > B + 2\Delta - 1$:

- Any option set containing a schedule edge **increases** potential by at most $2B + 3\Delta$
- Case 2: no message moved

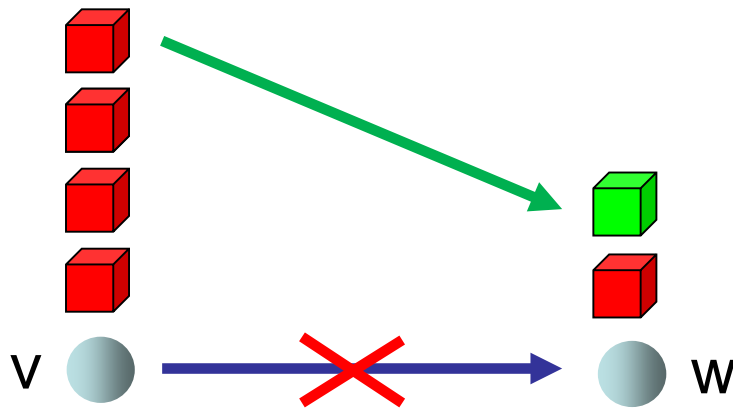


If loser at w , then switch roles, which increases potential by $\leq 2B + 3\Delta$

Anycast

If $T > B + 2\Delta - 1$:

- Any option set containing a schedule edge **increases** potential by at most $2B + 3\Delta$
- Case 2: no message moved

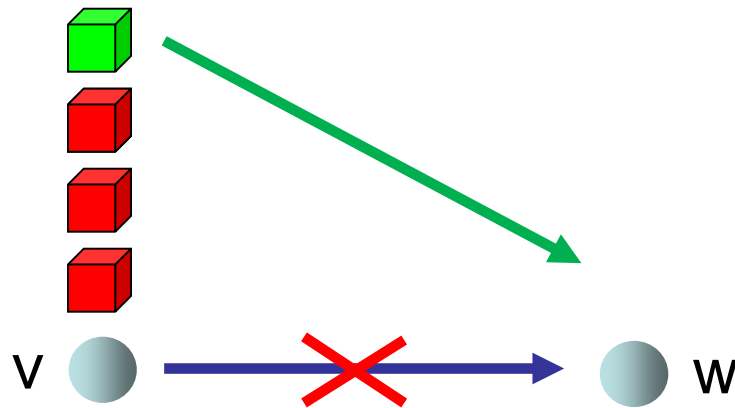


If loser at w , then switch roles, which increases potential by $\leq 2B + 3\Delta$

Anycast

If $T > B + 2\Delta - 1$:

- Any option set containing a schedule edge **increases** potential by at most $2B + 3\Delta$
- Case 2: no message forwarded

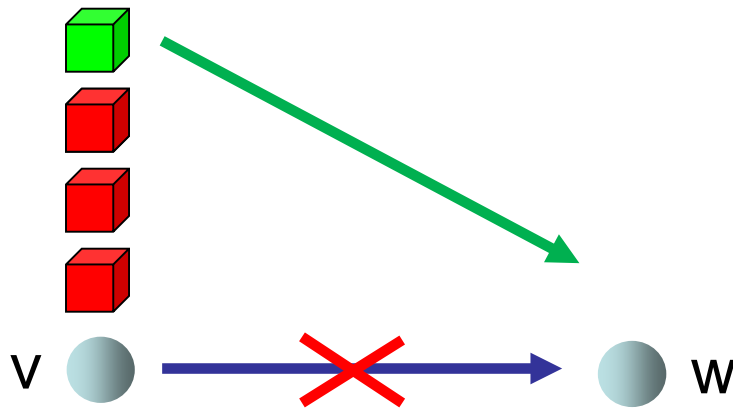


If no loser at w , then convert representative into loser, which also increases potential by $\leq 2B + 3\Delta$

Anycast

If $T > B + 2\Delta - 1$:

- Any option set containing a schedule edge **increases** potential by at most $2B + 3\Delta$
- Case 2: no message forwarded

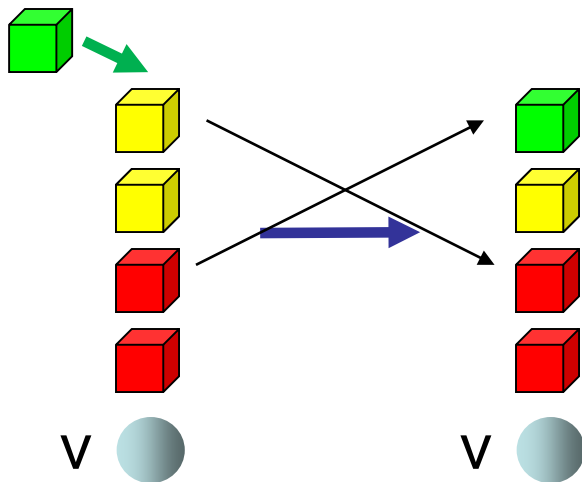


Exercise: argue, why bound is correct for case 2.

Anycast

If $T > B + 2\Delta - 1$:

- Deletion of injected message (with OPT-schedule) **decreases** potential by $H-B$

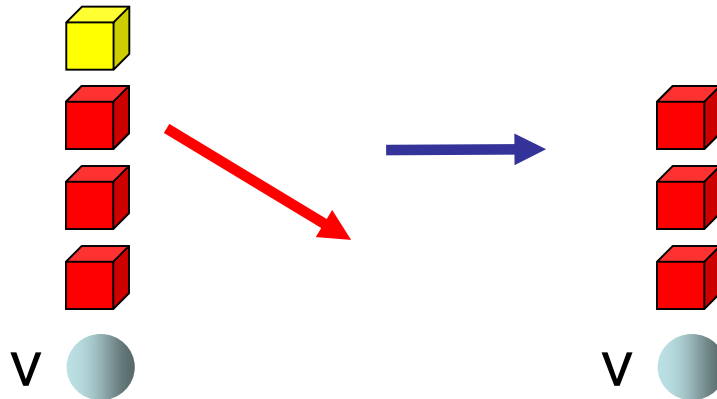


The highest loser is converted into the representative of the schedule and switches its role with the highest zombie (which is charged on the next slide).

Anycast

If $T > B + 2\Delta - 1$:

- Zombie **increases** potential by at most $H - B$
- Happens if zombie is converted to loser (which happens, e.g., if a loser is moved out of v).



Anycast

If $T > B + 2\Delta - 1$:

- Any option set not containing a schedule edge in OPT does **not increase** potential
- Any option set containing a schedule edge **increases** potential by at most $2B + 3\Delta$
- Deletion of injected message **decreases** potential by $H - B$
- Zombie **increases** potential by at most $H - B$

Anycast

Putting it all together:

- p : # option sets with schedule edge
- z : # injected messages without schedule
- d : # deleted messages
- Potential Φ :

$$\Phi < p(2B+3\Delta) + z(H-B) - d(H-B)$$

- s : # injected messages with schedule
 $p=L \cdot s$ (L : average path length)

Anycast

Hence,

$$d < \frac{p(2B+3\Delta)}{H-B} + z$$

Thus, number of messages delivered by balancing algo is at least

$$(s+z) - \left(s \frac{L(2B+3\Delta)}{H-B} + z \right) - H \cdot N$$

$H = \Omega(L(B+\Delta)/\varepsilon)$: $\sim(1-\varepsilon)s$ mgs delivered

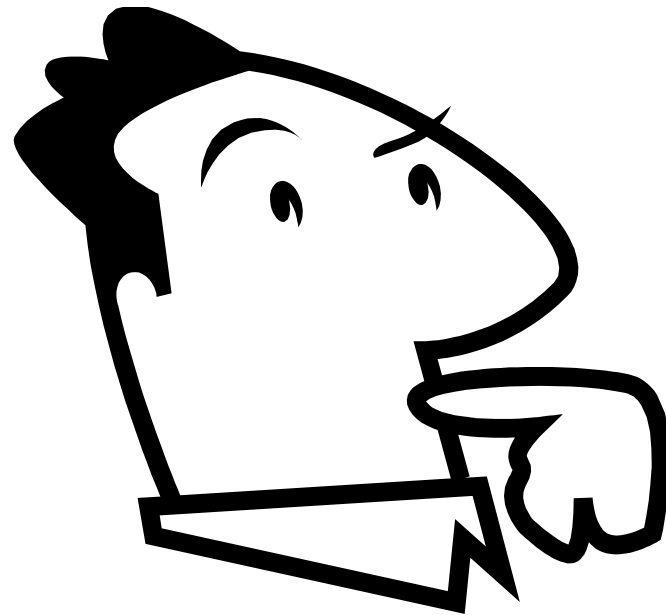
Anycast

Our context: clique, in which connections are chosen at random (instead of by an adversary)

- It should be expected that OPT uses short path lengths (i.e., $L=O(\log n)$). **Why?**
- Buffer size needed is only by a logarithmic factor higher than in OPT.

Anycast - Improvements

- Combine load balancing with broadcasting as a second priority (one primary and many secondary copies). Primary copies are dealt with via load balancing and have priority if a movement is allowed in the balancing algorithm. Once a copy makes it to the destination, it needs to invalidate the other copies (which is doable if a broadcast tree is maintained from the source).
- On top of a given routing solution, use caching (with a LRU (least-recently-used) eviction strategy, for example) to store anycast information so that it is easier to find processes belonging to an anycast predicate.



Questions?