# MultiSkipGraph

## A Self-stabilizing Overlay Network that Maintains Monotonic Searchability

Linghui Luo, Christian Scheideler and Thim Strothmann

Department of Computer Science
Paderborn University

May 15, 2019

# Outline

1. Introduction
   - Model
   - Problem Statement
2. Limitations
3. Two Protocols
   - MultiSkipGraph
   - MultiSkipGraph*
4. Demo
5. Experimental Comparison
   - Comparison in Stabilization
   - Comparison in Searchability
6. Summary

# Outline

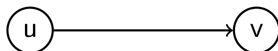
1. Introduction
   - Model
   - Problem Statement
2. Limitations
3. Two Protocols
   - MultiSkipGraph
   - MultiSkipGraph*
4. Demo
5. Experimental Comparison
   - Comparison in Stabilization
   - Comparison in Searchability
6. Summary

# Assumptions

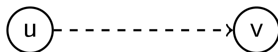
- Asynchronous distributed system
- Nodes communicate via message-passing
- Each node has:
  - a reference – $u$
  - a unique id – $u.id$
  - a channel – $u.channel$
- A node $u$ can send messages to a node $v$ if $u$ has the reference of $v$
- Each node $u$ initiates $\textsc{Search}(u, wID)$ messages

# Channel Connectivity Multigraph

- Graph $G = (V, E_e \cup E_i)$,
  - $V$: set of all nodes
  - $E_e$: set of all explicit edges
  - $E_i$: set of all implicit edges.
- Explicit Edge $(u, v)$: If node $u$ stores the reference of node $v$ locally.



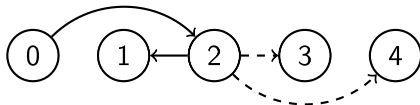

- Implicit Edge $(u, v)$: If there is a message in $u.channel$ containing the reference of node $v$.



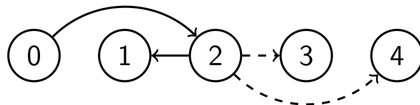
# Outline

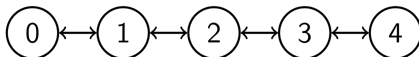
1. Introduction
   - Model
   - Problem Statement
2. Limitations
3. Two Protocols
   - MultiSkipGraph
   - MultiSkipGraph*
4. Demo
5. Experimental Comparison
   - Comparison in Stabilization
   - Comparison in Searchability
6. Summary

# Topological Self-stabilization



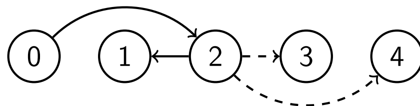
# Topological Self-stabilization

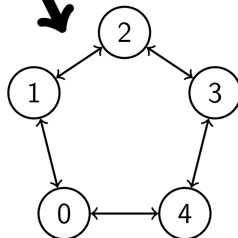


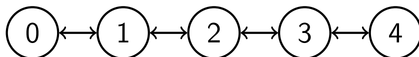



A line topology

# Topological Self-stabilization



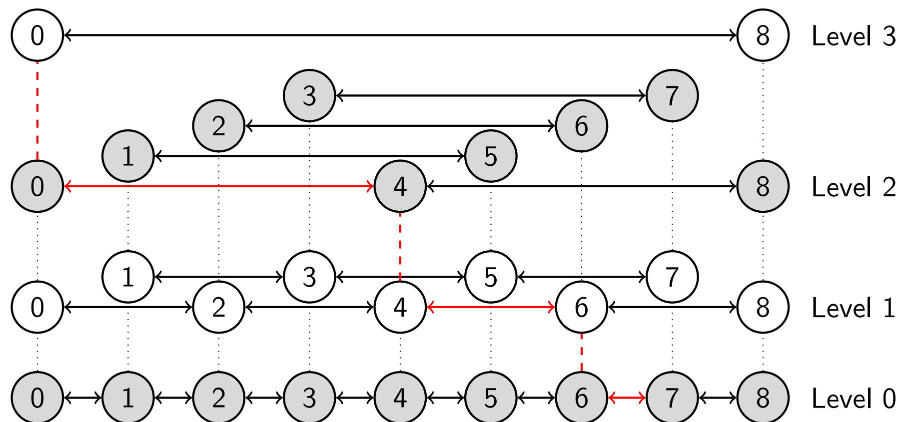



A line topology

A ring topology

# Perfect Skip Graph




Routing protocol: Greedy Search

# Monotonic Searchability

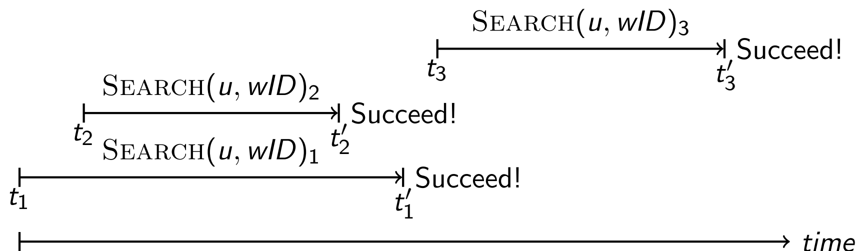



Figure: Example satisfies monotonic searchability

# Problem: Message Delivery

No protocol can **unconditionally** satisfy monotonic searchability, since messages are not necessarily delivered in FIFO order.

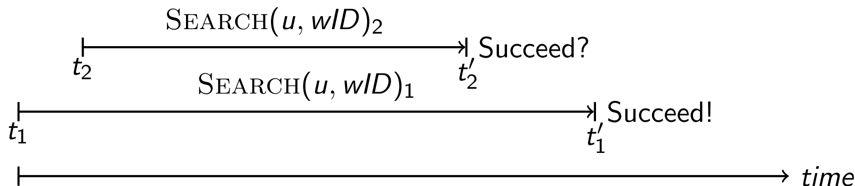



Figure: Messages initiated earlier could arrive later than messages initiated later

# Problem: Message Delivery

No protocol can **unconditionally** satisfy monotonic searchability, since messages are not necessarily delivered in FIFO order.

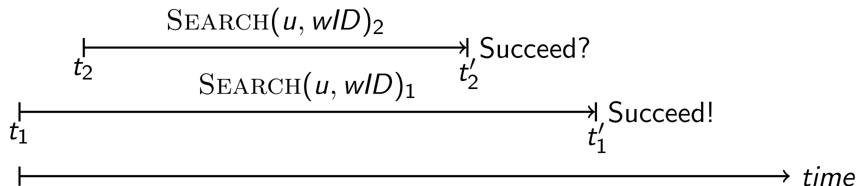
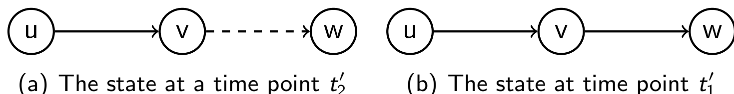


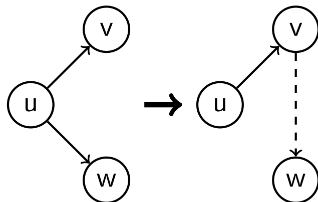

Figure: Messages initiated earlier could arrive later than messages initiated later




(a) The state at a time point $t'_2$ (b) The state at time point $t'_1$

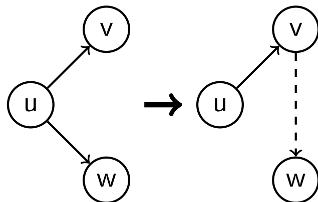
Figure: Example violates monotonic searchability

# Problem: Delegation



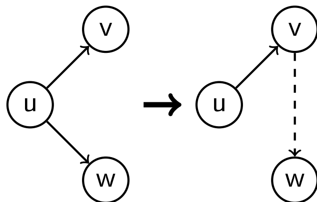

- Explicit edge $(u, w)$ is removed in simple delegation.

# Problem: Delegation



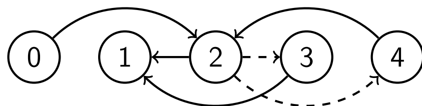

- Explicit edge $(u, w)$ is removed in simple delegation.
- After delegation node $w$ is not reachable from node $u$.

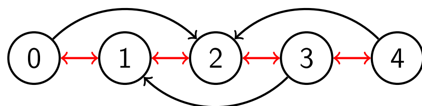
# Problem: Delegation




- Explicit edge $(u, w)$ is removed in simple delegation.
- After delegation node $w$ is not reachable from node $u$.
- A safe way: keep every explicit edge

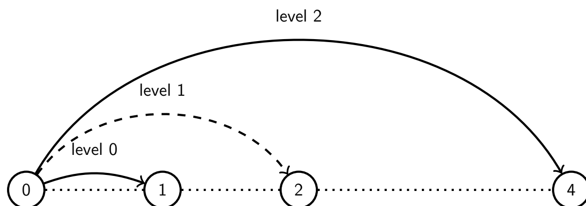
# Relaxed Self-stabilization



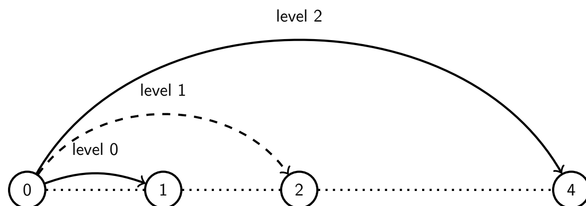



Supergraph of the line topology

# Limitations in Topology Maintenance

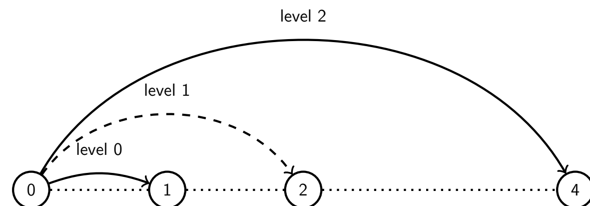


# Limitations in Topology Maintenance

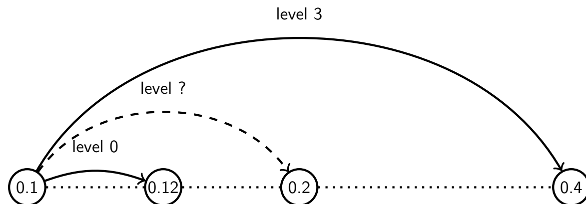



- Generally: the ids are not sequential!

# Limitations in Topology Maintenance

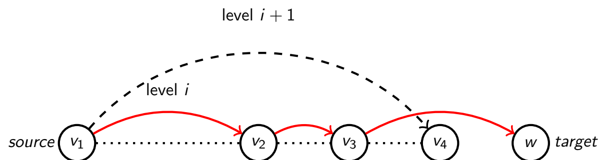



- Generally: the ids are not sequential!
- Perfect skip graph is not locally checkable.

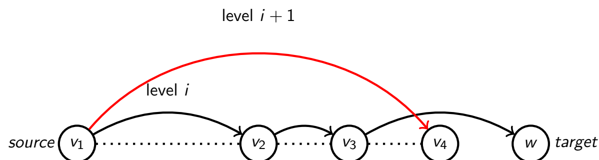


# Limitations in Monotonic Searchability

Greedy Search can not guarantee monotonic searchability




(a) First search path $v_1 \to v_2 \to v_3 \to w$




(b) Second search path $v_1 \to v_4 \to abort$

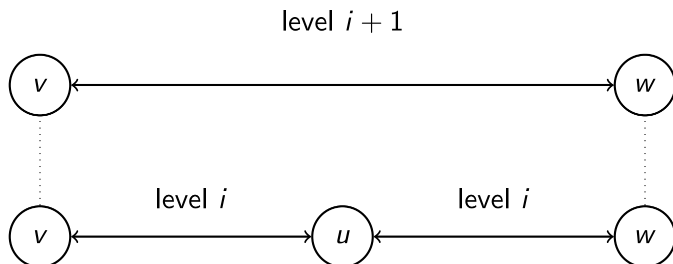
Figure: Example violates monotonic searchability

# Outline


1. Introduction
   - Model
   - Problem Statement
2. Limitations
3. Two Protocols
   - MultiSkipGraph
   - MultiSkipGraph*
4. Demo
5. Experimental Comparison
   - Comparison in Stabilization
   - Comparison in Searchability
6. Summary

# Observation

Bottom-up process: higher levels are built on top of lower levels.

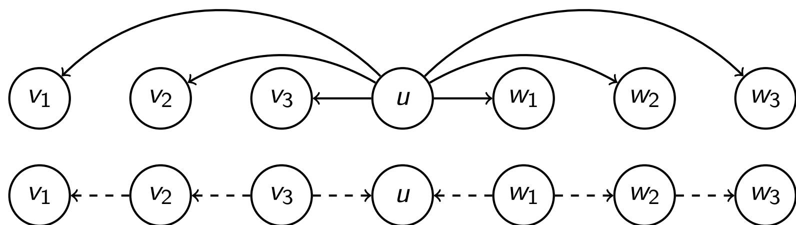


# MultiSkipGraph

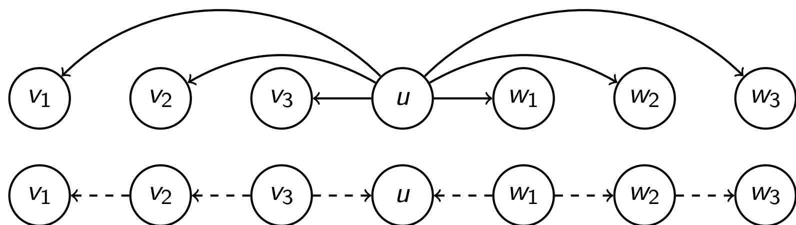
Periodically call $\text{Timeout}()$ action:

- the self-stabilizing part for topology maintenance
- the HybridSearch part for handling search requests

- Timeout(): Linear introduction by sending Introduce($*$) messages

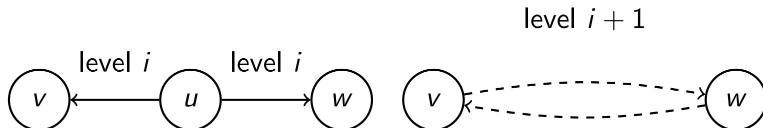


- TIMEOUT(): Linear introduction by sending INTRODUCE($*$) messages

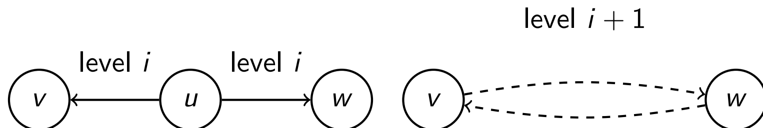



- INTRODUCE($v$):
  - Keep the closest neighbors as level-0 neighbors
  - Mark the old level-0 neighbors as *unknown*
  - Keep the references to *unknown* neighbors.

- Timeout(): Introduce level-$i$ neighbors for level-$i+1$ by sending IntroLevelNode($*$, $i+1$) messages for each level $i$



- Timeout(): Introduce level-$i$ neighbors for level-$i+1$ by sending IntroLevelNode($*$, $i+1$) messages for each level $i$

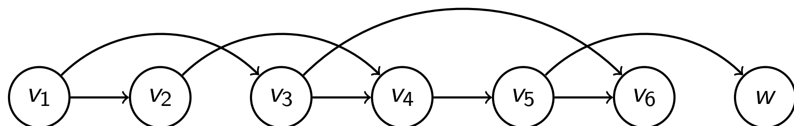



- IntroLevelNode($v$, $i$):
  - Always believe the newest information is correct
  - Mark old level-$i$ neighbor as *unknown*
  - Keep the references to *unknown* neighbors

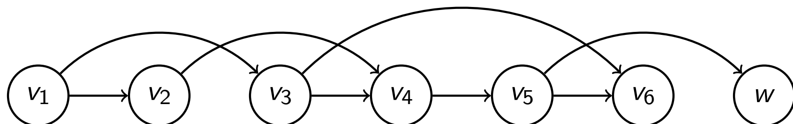
# MultiSkipGraph

## The HybridSearch part

SEARCH($v_1$, $wID$)

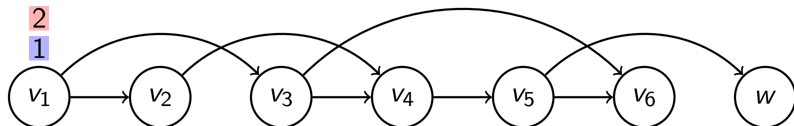


Search($v_1$, $wID$)



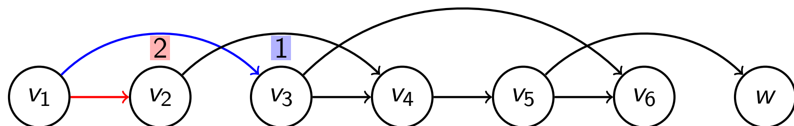

- $v_1$ doesn't forward Search($v_1$, $wID$) immediately, but buffers it.
- Timeout(): $v_1$ sends GreedyProbe() and GenericProbe() messages to itself to start two parallel probing processes.

Search($v_1$, $wID$)



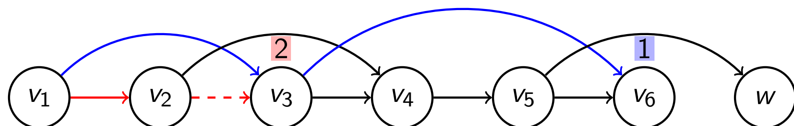

- 1 GreedyProbe($v_1$, $wID$, $seq$)
  - Node $v_1$ forwards it to $v_3$
- 2 GenericProbe($v_1$, $wID$, $Next$, $seq$), $Next = \{v_1\}$
  - Node $v_1$ updates $Next' = \{v_2, v_3\}$
  - Node $v_1$ forwards GenericProbe($v_1$, $wID$, $Next'$, $seq$) to $v_2$.

SEARCH($v_1$, $wID$)



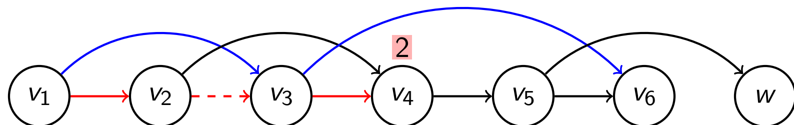

- 1 GREEDYPROBE($v_1$, $wID$, $seq$)
  - Node $v_3$ forwards it to $v_6$
- 2 GENERICPROBE($v_1$, $wID$, $Next$, $seq$), $Next = \{v_2, v_3\}$
  - Node $v_2$ updates $Next' = \{v_3, v_4\}$
  - Node $v_2$ forwards GENERICPROBE($v_1$, $wID$, $Next'$, $seq$) to $v_3$.

SEARCH($v_1$, $wID$)
SEARCH($v_1$, $wID$)



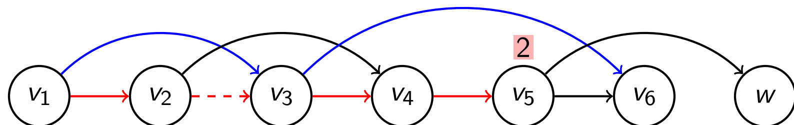

- New SEARCH($v_1$, $wID$) is initiated, buffering it
- 1 GREEDYPROBE($v_1$, $wID$, $seq$)
  - Node $v_6$ can't forward it anymore, greedy probe process failed.
- 2 GENERICPROBE($v_1$, $wID$, $Next$, $seq$), $Next = \{v_3, v_4\}$
  - Node $v_3$ updates $Next' = \{v_4, v_6\}$
  - Node $v_3$ forwards GENERICPROBE($v_1$, $wID$, $Next'$, $seq$) to $v_4$.

$\text{SEARCH}(v_1, wID)$
$\text{SEARCH}(v_1, wID)$



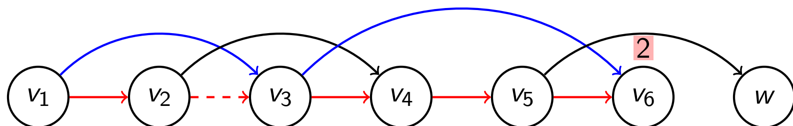

- 1 $\text{GREEDYPROBE}(v_1, wID, seq)$ failed
- 2 $\text{GENERICPROBE}(v_1, wID, Next, seq)$, $Next = \{v_4, v_6\}$
  - Node $v_4$ updates $Next' = \{v_5, v_6\}$
  - Node $v_4$ forwards $\text{GENERICPROBE}(v_1, wID, Next', seq)$ to $v_5$.

SEARCH($v_1$, $wID$)
SEARCH($v_1$, $wID$)




- 1 GREEDYPROBE($v_1$, $wID$, $seq$) failed
- 2 GENERICPROBE($v_1$, $wID$, $Next$, $seq$), $Next = \{v_5, v_6\}$
  - Node $v_5$ updates $Next' = \{v_6, w\}$
  - Node $v_5$ forwards GENERICPROBE($v_1$, $wID$, $Next'$, $seq$) to $v_6$.

SEARCH($v_1$, $wID$)
SEARCH($v_1$, $wID$)

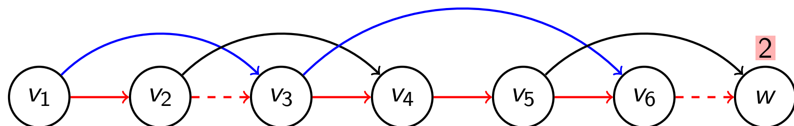



- 1 GREEDYPROBE($v_1$, $wID$, $seq$) failed
- 2 GENERICPROBE($v_1$, $wID$, $Next$, $seq$), $Next = \{v_6, w\}$
  - Node $v_6$ updates $Next' = \{w\}$
  - Node $v_6$ forwards GENERICPROBE($v_1$, $wID$, $Next'$, $seq$) to $w$.

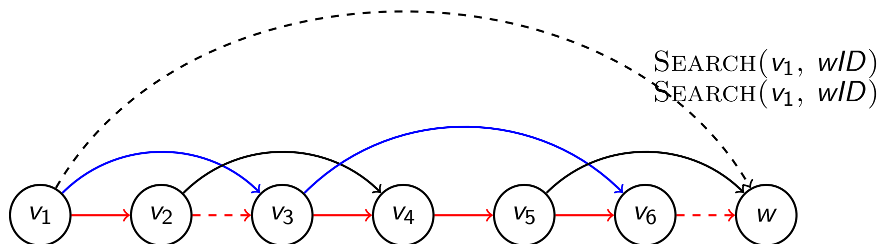
# MultiSkipGraph

## The HybridSearch part

Search($v_1$, $wID$)
Search($v_1$, $wID$)




- 1 GreedyProbe($v_1$, $wID$, $seq$) failed
- 2 GenericProbe($v_1$, $wID$, $Next$, $seq$), $Next = \{w\}$
  - Succeed!
  - Node $w$ sends ProbeSuccess($wID$, $seq$, $w$) back to $v_1$.




- 1 GreedyProbe($v_1$, $wID$, $seq$) failed
- 2 GenericProbe($v_1$, $wID$, $Next$, $seq$), $Next = \{w\}$
  - Succeed! Node $w$ sends ProbeSuccess($wID$, $seq$, $w$) back to $v_1$.
  - Node $v_1$ sends all Search($v_1$, $wID$) to $w$.

How is monotonic searchability maintained?

How is monotonic searchability maintained?

- $\textsc{Search}(v, wID)$ messages are only sent to target node $w$ when $v$ receives $\textsc{ProbeSuccess}(wID, seq, w)$ message.

How is monotonic searchability maintained?

- SEARCH($v$, $wID$) messages are only sent to target node $w$ when $v$ receives PROBESUCCESS($wID$, $seq$, $w$) message.
- Node $v$ receives PROBESUCCESS($wID$, $seq$, $w$) messages: there is a directed path from $v$ to $w$.

How is monotonic searchability maintained?

- Search($v$, $wID$) messages are only sent to target node $w$ when $v$ receives ProbeSuccess($wID$, $seq$, $w$) message.
- Node $v$ receives ProbeSuccess($wID$, $seq$, $w$) messages: there is a directed path from $v$ to $w$.
- MultiSkipGraph removes no explicit edge: the directed path from $v$ to $w$ always exists.

How is monotonic searchability maintained?

- Search($v$, $wID$) messages are only sent to target node $w$ when $v$ receives ProbeSuccess($wID$, $seq$, $w$) message.
- Node $v$ receives ProbeSuccess($wID$, $seq$, $w$) messages: there is a directed path from $v$ to $w$.
- MultiSkipGraph removes no explicit edge: the directed path from $v$ to $w$ always exists.
- Probing for Search($v$, $wID$) will always succeeds.

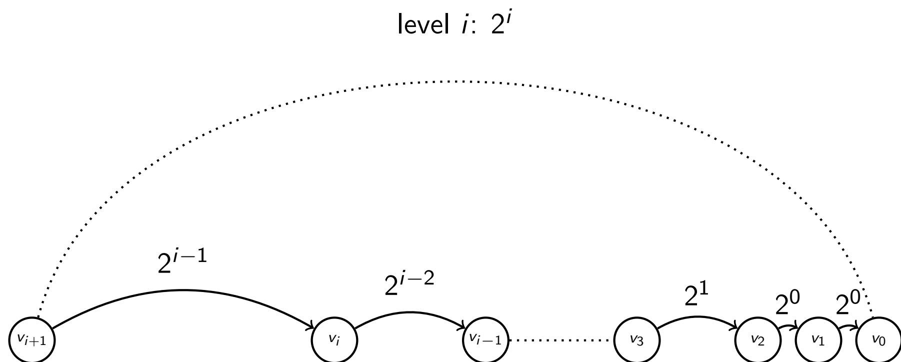
# Outline


1. Introduction
   - Model
   - Problem Statement
2. Limitations
3. Two Protocols
   - MultiSkipGraph
   - MultiSkipGraph*
4. Demo
5. Experimental Comparison
   - Comparison in Stabilization
   - Comparison in Searchability
6. Summary

# Observation

## Deterministic Search Path

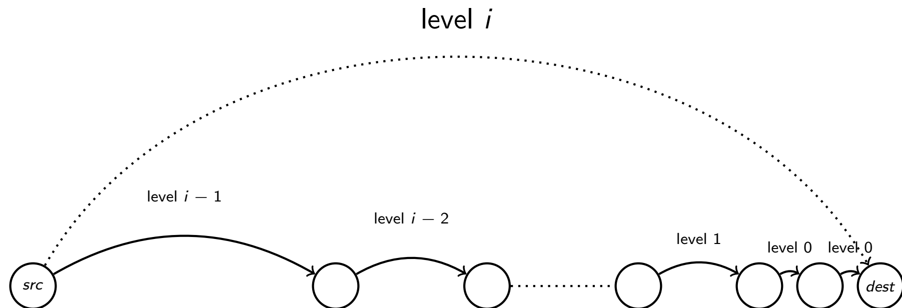
Nodes $v_{i+1}$ and $v_0$ are level-$i$ neighbors



# MultiSkipGraph*

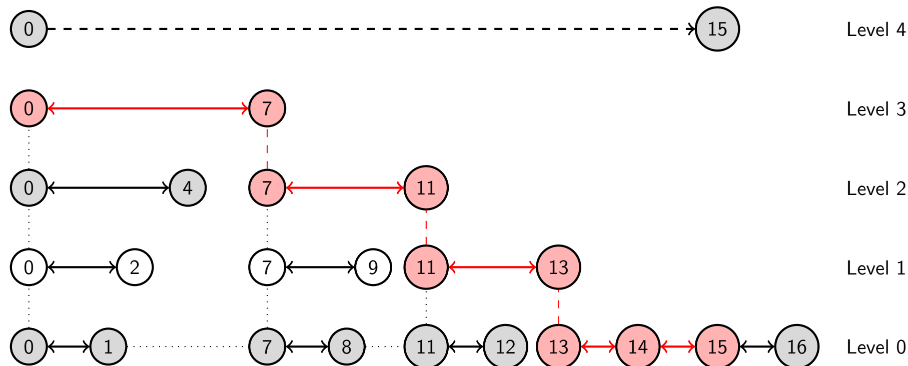
## Changes in the self-stabilizing part

- More strict in updating level neighbors
- IntroLevelNode(*dest*, *i*): Probing the deterministic search path by forwarding ProbeLevelNode(*src*, *dest*, *i*, *j*) message



# Observation

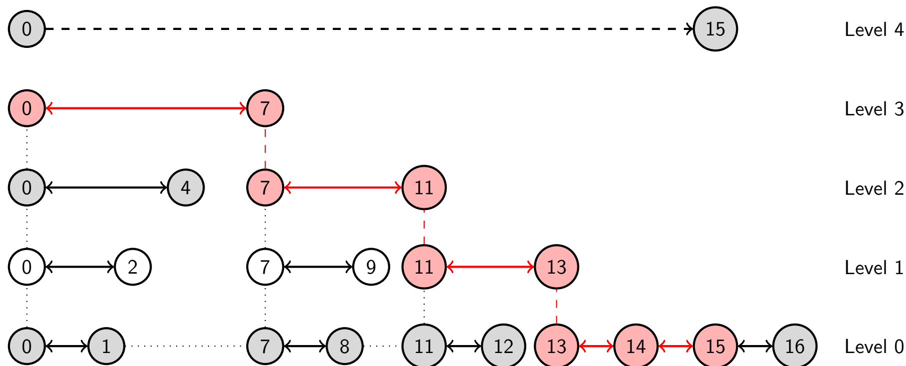
## Deterministic Search Path




- Deterministic Search Path exists, but nodes 1 and 15 are not level-4 neighbors

# Observation

## Deterministic Search Path




- Deterministic Search Path exists, but nodes 1 and 15 are not level-4 neighbors
- Still needs to update every time when probe succeeds

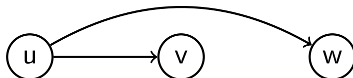
TIMEOUT(): Safely delegate the *unknown* neighbors

# MultiSkipGraph*

## Changes in the self-stabilizing part

Timeout(): Safely delegate the *unknown* neighbors

- $u$ wants to delegate its right *unknown* neighbor $w$.

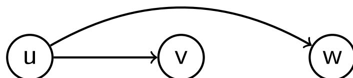


# MultiSkipGraph*

## Changes in the self-stabilizing part

Timeout(): Safely delegate the *unknown* neighbors

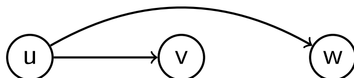
- $u$ wants to delegate its right *unknown* neighbor $w$.



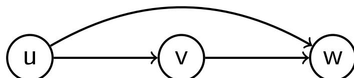

- $u$ sends SafeIntroduce($w$, $u$) to its right neighbor $v$ which is closest to $w$ and $v.id < w.id$.

Timeout(): Safely delegate the *unknown* neighbors

- $u$ wants to delegate its right *unknown* neighbor $w$.

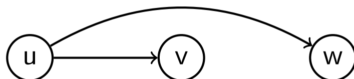



- $u$ sends SafeIntroduce$(w, u)$ to its right neighbor $v$ which is closest to $w$ and $v.id < w.id$.
- $v$ adds $w$ as its *unknown* neighbor and sends SafeDeletion$(w)$ back to $u$.

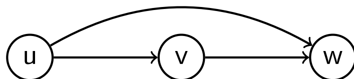


TIMEOUT(): Safely delegate the *unknown* neighbors

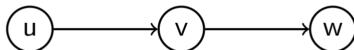
- $u$ wants to delegate its right *unknown* neighbor $w$.



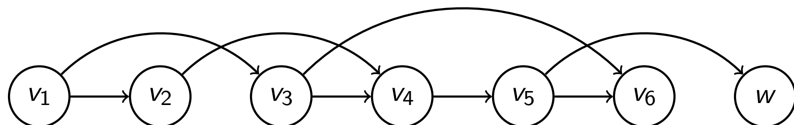

- $u$ sends SAFEINTRODUCE($w$, $u$) to its right neighbor $v$ which is closest to $w$ and $v.id < w.id$.
- $v$ adds $w$ as its *unknown* neighbor and sends SAFEDELETION($w$) back to $u$.




- $u$ deletes $w$.

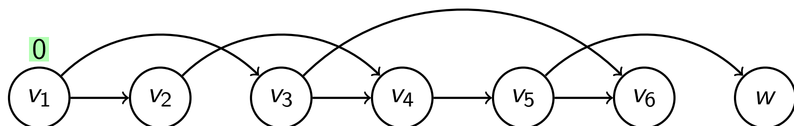


Search($v_1$, $wID$)



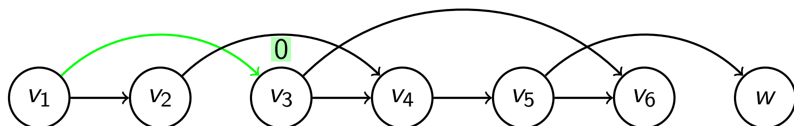

- Timeout(): $v_1$ sends a SlowGreedyProbe() message to itself to start a probing process.

Search($v_1$, $wID$)



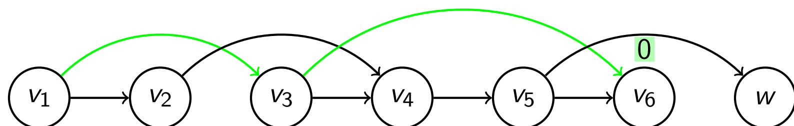

- 0 SlowGreedyProbe($v_1$, $wID$, $Prev$, $Next$, $seq$), $Prev = \emptyset$, $Next = \{v_1\}$
  - Node $v_1$ updates $Prev' = \{v_1\}$ and $Next' = \{v_2, v_3\}$
  - Node $v_1$ forwards SlowGreedyProbe($v_1$, $wID$, $Prev'$, $Next'$, $seq$) to $v_3$

Search($v_1$, $wID$)



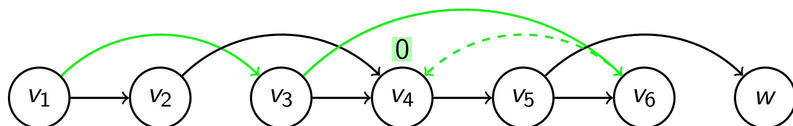

- 0 SlowGreedyProbe($v_1$, $wID$, $Prev$, $Next$, $seq$), $Prev = \{v_1\}$, $Next = \{v_2, v_3\}$
  - Node $v_3$ updates $Prev' = \{v_1, v_3\}$ and $Next' = \{v_2, v_4, v_6\}$
  - Node $v_3$ forwards SlowGreedyProbe($v_1$, $wID$, $Prev'$, $Next'$, $seq$) to $v_6$

SEARCH($v_1$, $wID$)



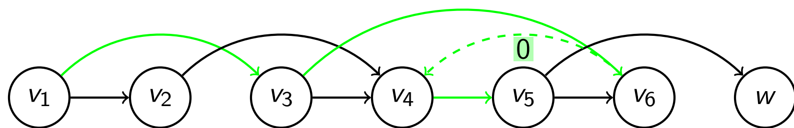

- 0 SLOWGREEDYPROBE($v_1$, $wID$, $Prev$, $Next$, $seq$), $Prev = \{v_1, v_3\}$, $Next = \{v_2, v_4, v_6\}$
  - Node $v_6$ updates $Prev' = \{v_1, v_3, v_6\}$ and $Next' = \{v_2, v_4\}$
  - Node $v_6$ forwards SLOWGREEDYPROBE($v_1$, $wID$, $Prev'$, $Next'$, $seq$) to $v_4$

Search($v_1$, $wID$)



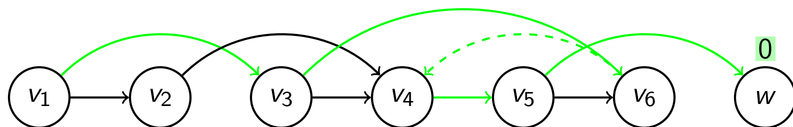

- 0 SlowGreedyProbe($v_1$, $wID$, $Prev$, $Next$, $seq$), $Prev = \{v_1, v_3, v_6\}$ , $Next = \{v_2, v_4\}$
  - Node $v_4$ updates $Prev' = \{v_1, v_3, v_6, v_4\}$ and $Next' = \{v_2, v_5\}$
  - Node $v_4$ forwards SlowGreedyProbe($v_1$, $wID$, $Prev'$, $Next'$, $seq$) to $v_5$

Search($v_1$, $wID$)




- 0 SlowGreedyProbe($v_1$, $wID$, $Prev$, $Next$, $seq$), $Prev = \{v_1, v_3, v_6, v_4\}$, $Next = \{v_2, v_5\}$
  - Node $v_5$ updates $Prev' = \{v_1, v_3, v_6, v_4, v_5\}$ and $Next' = \{v_2, w\}$
  - Node $v_5$ forwards SlowGreedyProbe($v_1$, $wID$, $Prev'$, $Next'$, $seq$) to $w$

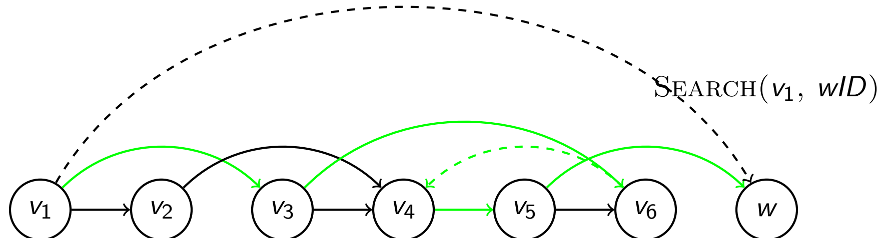
SEARCH($v_1$, $wID$)




- 0 SLOWGREEDYPROBE($v_1$, $wID$, $Prev$, $Next$, $seq$), $Prev = \{v_1, v_3, v_6, v_4, v_5\}$, $Next = \{v_2, w\}$
  - Succeed!
  - Node $w$ sends PROBESUCCESS($wID$, $seq$, $w$) back to $v_1$.

# MultiSkipGraph*

## The SlowGreedySearch part




- 0 SlowGreedyProbe($v_1$, $wID$, $Prev$, $Next$, $seq$), $Prev = \{v_1, v_3, v_6, v_4, v_5\}$, $Next = \{v_2, w\}$
  - Succeed! Node $w$ sends ProbeSuccess($wID$, $seq$, $w$) back to $v_1$.
  - Node $v_1$ sends Search($v_1$, $wID$) to $w$.

# Demo

https://www.youtube.com/watch?v=S8yd7fApSfk&list=PLSdezCrzv5YPpamQYw46HK-uNdyhyQVbt

# Outline

1 Introduction
- Model
- Problem Statement

2 Limitations

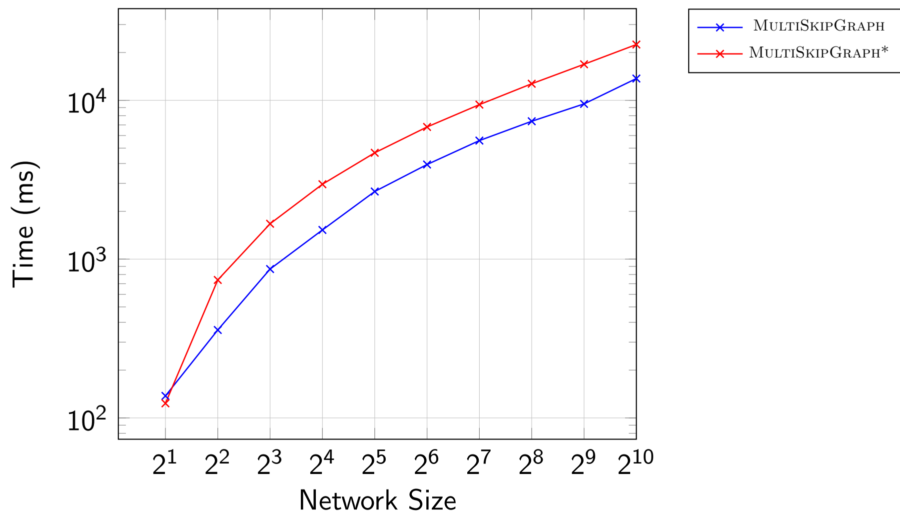
3 Two Protocols
- MultiSkipGraph
- MultiSkipGraph*

4 Demo

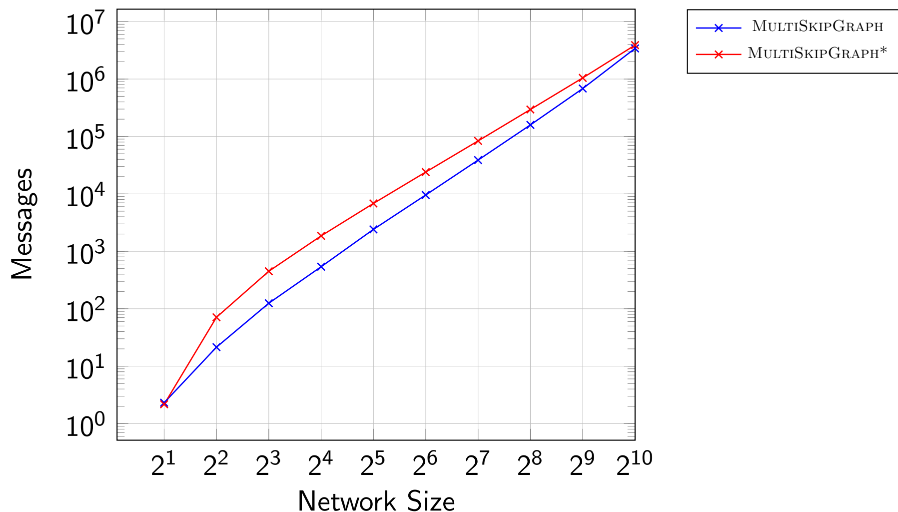
5 Experimental Comparison
- Comparison in Stabilization
- Comparison in Searchability

6 Summary

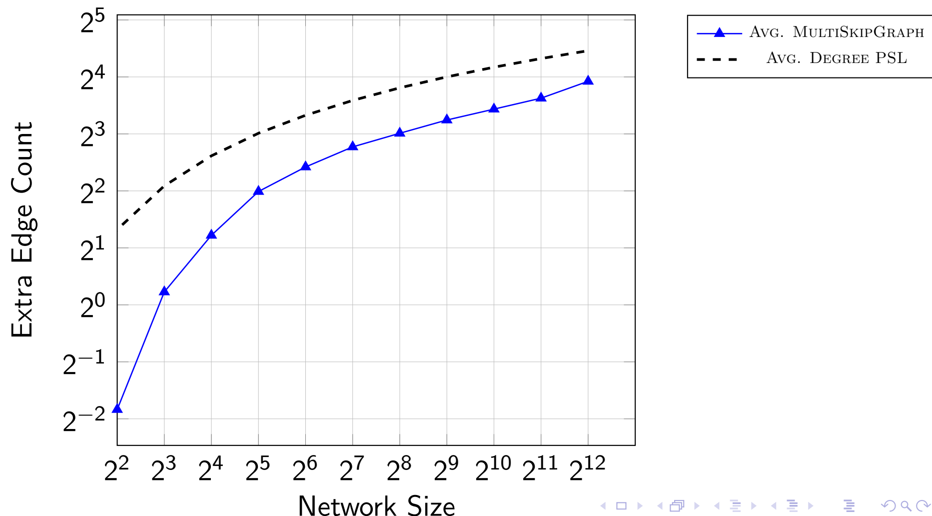
# Comparison in Stabilization



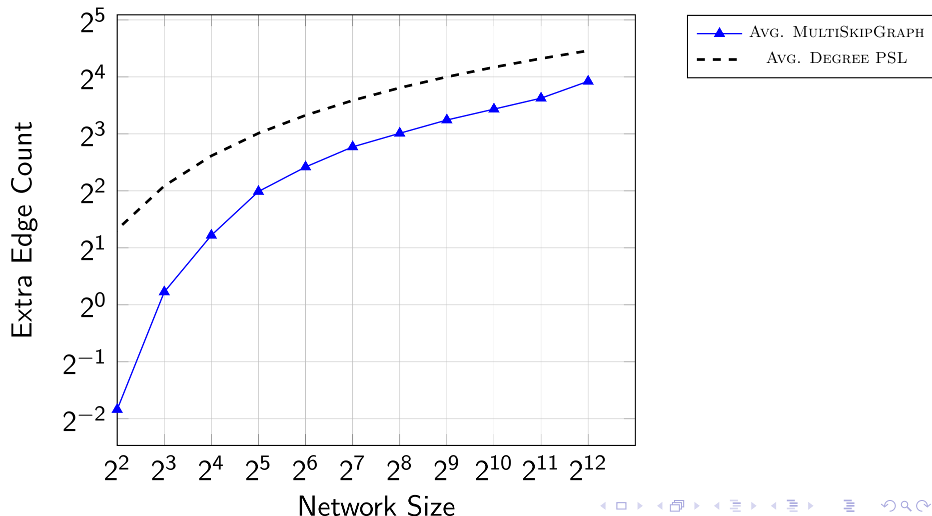
# Comparison in Stabilization



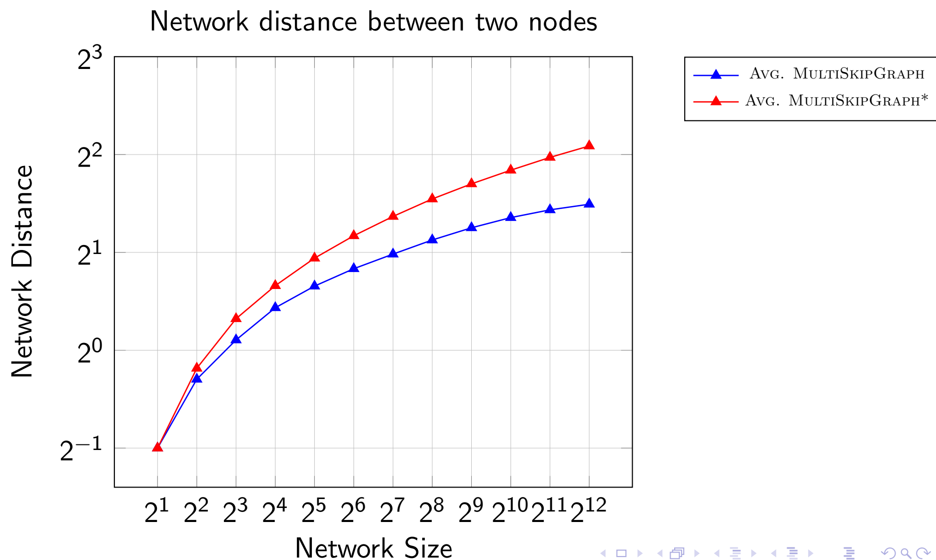
# Comparison in Stabilization



# Comparison in Stabilization



# Comparison in Stabilization



# Outline

1 Introduction
- Model
- Problem Statement

2 Limitations

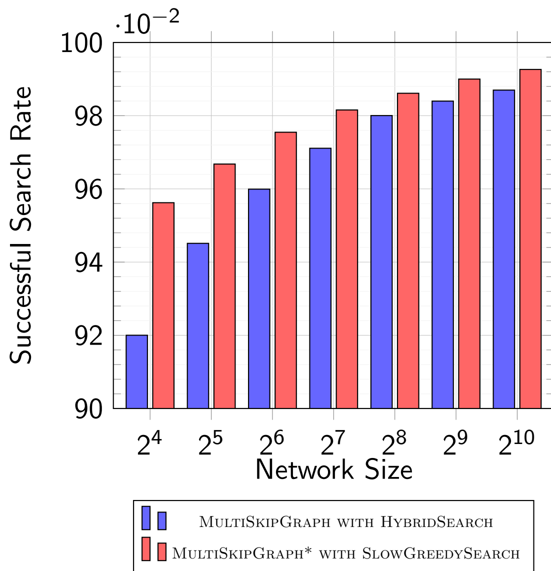
3 Two Protocols
- MultiSkipGraph
- MultiSkipGraph*

4 Demo

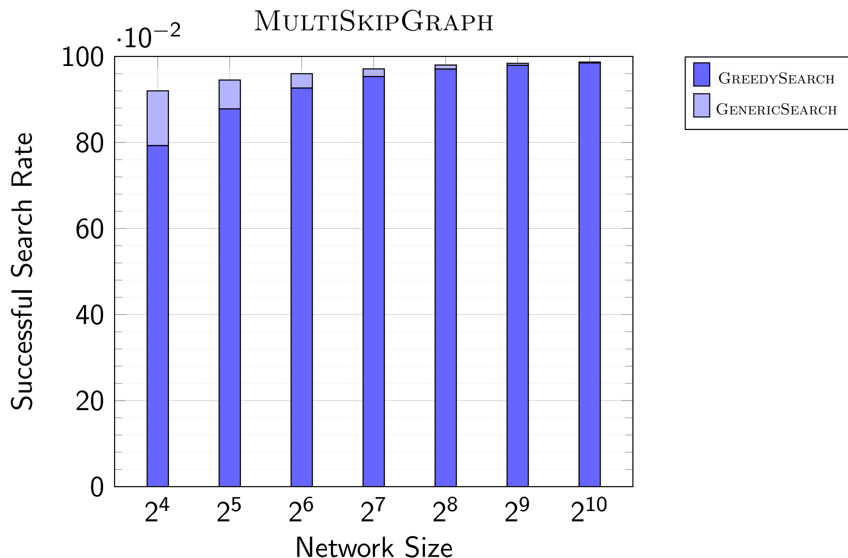
5 Experimental Comparison
- Comparison in Stabilization
- Comparison in Searchability

6 Summary

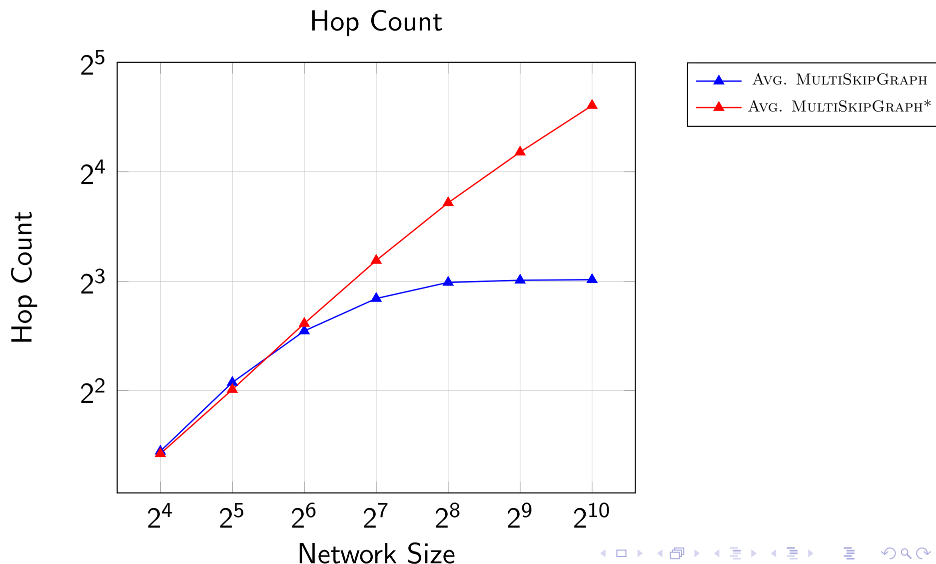
# Comparison in Searchability



# Comparison in Searchability



# Comparison in Searchability



# Summary

- MultiSkipGraph
  - Fast
  - Final topology: supergraph of the perfect skip graph
  - Extra local memory overhead
  - Reduced search time
- MultiSkipGraph*
  - Final topology: exactly the perfect skip graph
  - More messages
  - Higher successful search rate with SlowGreedySearch
- Outlook
  - Churn management – node insertion and deletion
  - Extend to circular skip graph for more robustness
  - Experiments on multiple computers

# Primitives

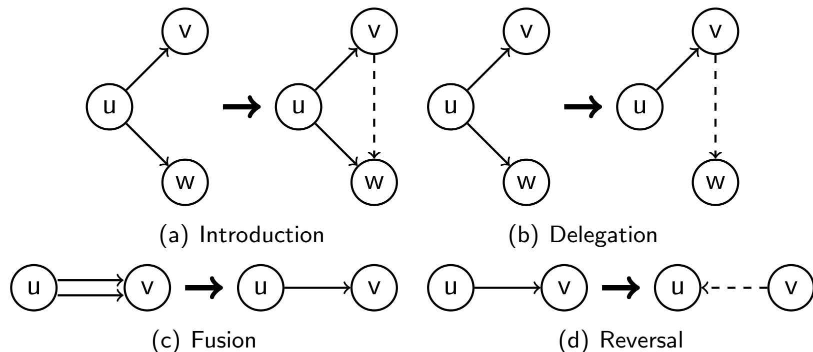
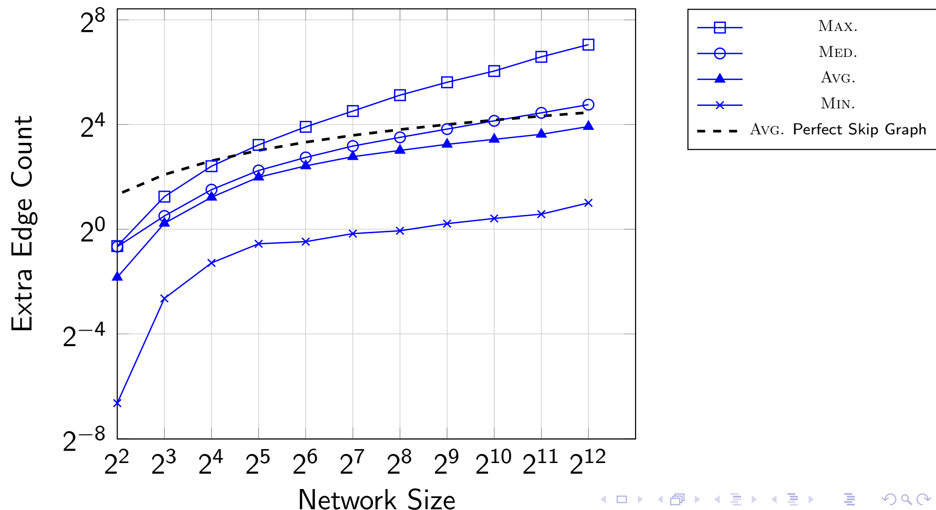



Figure: Illustration of four primitives

- *LeftLevel*[i]: the left level-$i$ neighbor
- *RightLevel*[i]: the right level-$i$ neighbor
- *LeftUnknown*: the set of left neighbors which are not assigned to any level, marked as *unknown*.
- *RightUnknown*: the set of right neighbors which are not assigned to any level, marked as *unknown*.
- *Left*: the set of all left neighbors
- *Right*: the set of all right neighbors

# Comparison in Stabilization



# Comparison in Stabilization

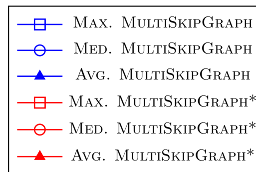
Network distance between two nodes

Network Distance

$2^3$
$2^2$
$2^1$
$2^0$
$2^{-1}$

$2^1$ $2^2$ $2^3$ $2^4$ $2^5$ $2^6$ $2^7$ $2^8$ $2^9$ $2^{10}$ $2^{11}$ $2^{12}$

Network Size



# Comparison in Searchability