A Randomized Algorithm for Label Assignment in Dynamic Networks

Meg Walraed-Sullivan · **Radhika Niranjan Mysore** · **Keith Marzullo** · **Amin Vahdat**

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California 92093-0404, USA
E-mail: megwalraedsullivan@gmail.com, radhika@cs.ucsd.edu, marzullo@cs.ucsd.edu, vahdat@cs.ucsd.edu

Abstract A basic problem in distributed computing has to do with assigning unique labels — that is, names or addresses — to network elements. Some approaches to solving this problem include using static assignment (e.g., MAC addresses), or using a centralized authority (e.g., DHCP). In this paper, we present an approach that is suitable for dynamic environments: where the rules constraining the label choices depend on the network topology, which in turn can change. This problem arose in the context of automatic address assignment in large-scale data center networks, and so we consider issues such as the scalability of message load and convergence time. We give a new algorithm, called the Decider/Chooser Protocol, and show its use in the assignment of labels in data center networks. We evaluate the correctness of the Decider/Chooser Protocol through proofs and model checking, and explore its performance via mathematical analysis and simulation. Through this evaluation, we find that the Decider/Chooser Protocol is well-suited for label assignment in the data center environment.

Keywords Distributed Algorithms · Randomized Algorithms · Practical Protocols · Fault Tolerance · Data Center Networking

1 Introduction

The assignment of labels to network elements is a well-understood problem. Often, labels can be assigned statically, as with MAC addresses in traditional Layer 2 networks, or by a central authority as in DHCP in Layer 3 networks. When a dynamic, decentralized solution is required, one can employ a Consensus-based state machine approach [15]. However, dynamic assignment becomes more complex when the rules for labels depend on connectivity and when connectivity (and, hence, the labels) can change over time. As we will show in Appendix A, using a state machine approach becomes difficult in this case.

We came to this problem while designing ALIAS [19], a protocol that considers the problem of automatic label assignment in large-scale hierarchical data center networks. Practical constraints were important. We wished a decentralized solution because a centralized approach has its own challenges, such as exhibiting a single point of failure. Additionally, at the scale of the data center, establishing communication between a centralized component and all network elements necessitates either flooding or a separate out-of-band control network, an undesirable requirement. As well as being decentralized, our solution needed to scale to hundreds of thousands of nodes, and to be robust in the face of miswirings. It needed to have a low message overhead and convergence time, to be robust under transient startup conditions, and to retain high availability and quick stabilization after failures. Finally, a simple solution was ideal, since it was important that it be designed and implemented correctly. This paper describes a simple randomized approach that meets our practical goals.

We formally specify the problem of label assignment in Section 3 and provide a new algorithm, the Decider/Chooser Protocol (DCP), as a solution to this problem in Section 4. In Sections 5 and 6 we discuss the correctness and performance of DCP and provide a probabilistic analysis of its convergence time. In Section 7, we extend DCP to solve the issue of automatic labeling in data center networks, and in Section 8 we offer another application of DCP, handoff in wireless networks. Finally, we discuss context and related topics in Section 9.

2 ALIAS Details

In this section, we present a brief overview of ALIAS in order to help the reader to understand the concepts to follow.

In ALIAS, switches are organized into a multi-rooted tree, with end hosts connected to leaf switches, as shown in Figure 1. The ALIAS protocol includes three components: **Level Assignment**, **Label Assignment** and **Communication**. First, switches run a distributed protocol to determine their levels, L_1 through L_n, within the tree. They then select *labels* that will form the basis for communication. To select labels, switches first choose *coordinates*, which are values from a given domain. These coordinates are then concatenated along paths from the roots of the tree to switches in order to form switch labels. There may be multiple paths from the top level of the tree to any given switch, so switches in ALIAS can have multiple labels.[1] A host label is formed by concatenating a host h's neighboring L_1 switch l_1's labels to the number of the port on which h connects to l_1. Finally, once labels have been established, switches communicate with other switches and hosts using these labels as a basis for the ALIAS routing and forwarding protocols.

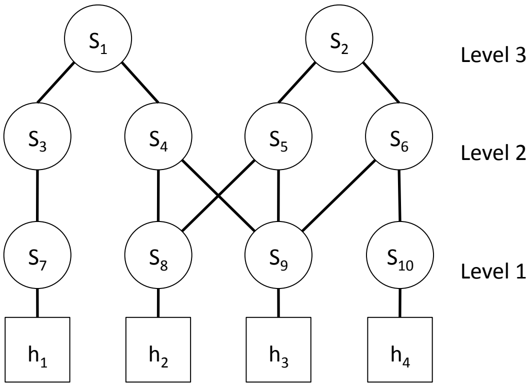

Fig. 1: ALIAS Topology

In this paper, we consider the problem of assigning coordinates to switches in ALIAS. In Section 3, we describe the requirements of coordinates and labels in order for ALIAS communication to function properly. We specify the Label Selection Problem and show how coordinate selection in ALIAS maps to this problem.

3 The Label Selection Problem

In the Label Selection Problem (LSP), we consider topologies made up of *chooser* processes connected to *decider* processes, as shown in Figure 2. These chooser and decider processes correspond to nodes at adjacent levels of a multi-rooted tree in ALIAS. All processes have globally unique identifiers, such as MAC addresses, chosen from a large address space. Desired is an assignment of labels from a small label space to choosers such that any two choosers that are connected to the same decider have distinct labels; this is the key requirement that allows ALIAS communication to operate over assigned labels.

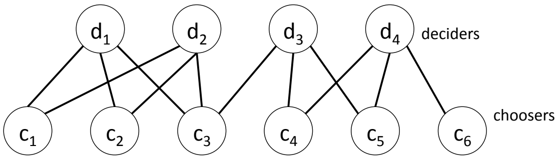

Fig. 2: Sample Label Selection Problem Topology

More formally, each chooser c has a set $c.deciders$ of deciders associated with it. We denote c's current choice of label with $c.me$, and $c.me = \perp$ indicates that c has not chosen a label.

A chooser c is connected to each decider in $c.deciders$ with a fair lossy link. Such links can drop messages, but if two processes p and q are connected by a fair lossy link and p sends m infinitely often to q, then q will receive m infinitely often.

Both decider and chooser processes can *crash* in a fail-stop manner (thus going from *up* to *down*) and can *recover* (thus going from *down* to *up*) at any time. We assume that a process writes its state to stable storage before sending a set of messages. When a process recovers, it is restored to the state that it was in before sending the last set of messages: duplicate messages may be sent upon recovery. So, we treat recovered processes as perhaps slow processes, and assume that duplicate messages can occur.

Figure 3 illustrates sets of choosers and the deciders they share, based on the topology shown in Figure 2. For instance, chooser c_3 shares deciders d_1 and d_2 with choosers c_1 and c_2 and shares decider d_3 with choosers c_4 and c_5. Because of this, c_3 may not select the same label as any of choosers c_1, c_2, c_4 and c_5. However, c_3 and c_6 are free to select the same label. In fact, the highlighted sub-graphs in Figure 3 correspond to the maximal bipartite graphs embedded in the topology.

---

[1] In Section 7, we show how ALIAS reduces the number of labels per host.

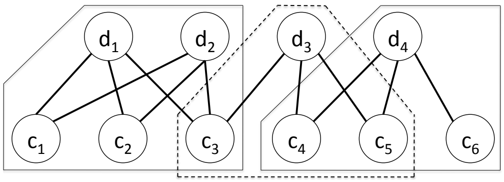

Fig. 3: Choosers and Shared Deciders

We more formally specify LSP with the following two properties:

Progress: For each chooser c, once c remains up, eventually $c.me \neq \bot$.

Distinctness: For each distinct pair of choosers c_1 and c_2, once c_1 and c_2 remain up and there is some decider that remains up and remains in $c_1.deciders \cap c_2.deciders$, eventually always $c_1.me \neq c_2.me$.

As specified, a chooser does not know when its choice satisfies **Distinctness**. Indeed, it is impossible for a chooser to know this without further constraining the problem. Consider the example in Figure 4, where nodes c_1 through c_3 are choosers and d_1 through d_4 are deciders. A valid set of choices is $c_1.me = c_3.me = 0$ and $c_2.me = 1$. If a link between c_3 and d_1 appears—perhaps it is newly added—then, this set of choices is no longer valid: c_1 and c_3 now share decider d_1 and so $c_1.me$ should differ from $c_3.me$. This could also occur were a new decider d_5 to appear that connects to both c_1 and c_3.

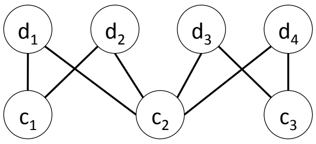

Fig. 4: Stability Example

Thus, if an application based on LSP requires a chooser to know that its label will not change, then one would need to ensure, for example, that new connections between deciders and choosers cannot be created.

4 The Decider/Chooser Protocol

One might be tempted to implement LSP with Consensus, because Consensus can be used to solve the arbitration problem in **Distinctness**. In Appendix A, we show that using Consensus presents considerable difficulties in the face of dynamic network environments and changing sets of deciders and choosers. Instead, we develop here the Decider/Chooser Protocol (DCP), which is a randomized protocol that solves LSP with dynamic sets of deciders and choosers. The input to DCP is a bipartite graph between a set of *choosers* and a set of *deciders*, and the output is an assignment of labels to choosers such that all choosers have non-$\bot$ labels and no two choosers sharing a decider have the same label.

DCP proceeds as follows: A chooser c repeatedly chooses a label *me* from some range of labels and sends it to $c.deciders$, its set of neighboring deciders. If a decider d has not currently assigned *me* to another chooser, then it assigns *me* to c. To accomplish this, d maintains a table $d.chosen$ of labels that it has accepted from choosers. If *me* is not in $d.chosen$ for some other chooser c', then d sets $d.chosen[c]$ to *me* and sends a reply to c indicating that *me* was accepted. Otherwise, d sets $d.chosen[c]$ to $\bot$ (indicating that d has not assigned a value for c) and sends a reply to c indicating that its choice was rejected. d includes the set of labels assigned to other choosers in this reply as hints so c can avoid them when choosing another label.

To guard against difficulties caused by message duplication and reordering, each chooser attaches a monotonically increasing sequence number with each choice that it sends to a decider. A deciders d keeps records in $d.last\_seq[c]$ of the largest sequence number seen from each chooser c and ignores messages from c with sequence numbers less than $d.last\_seq[c]$. This allows us to consider channels between choosers and deciders as fair lossy FIFO channels: if p sends m_1 to q and then sends m_2 to q, q may receive m_1, m_2, both, or neither of these messages, but once it receives m_2 it will never receive m_1.

Listing 1 gives the decider's state and its two Actions F and G. Action G was described in the previous paragraph; Action F executes when decider d first learns that it is connected to a new chooser c. When this happens, d updates its set $d.choosers$ of known choosers and initializes $d.chosen[c]$

Listing 1: Decider Algorithm

```
set⟨Chooser⟩ choosers = ...
Choice[choosers] chosen = all[⊥]
int[choosers] last_seq = all[0]

// when connected to new chooser c
F: when new chooser c
    choosers ← choosers ∪ {c}
    chosen[c] ← ⊥
    last_seq[c] ← 0

// respond to a message from chooser c
G: when receive ⟨s, x⟩ from c
    if s≥last_seq[c]
        last_seq[c] ← s
        if ∃ c' ∈ (choosers \ {c}): chosen[c'] == x
            chosen[c] ← ⊥
        else
            chosen[c] ← x
    hints ← {chosen[c']_{∀ c' ∈ (choosers \ {c})}} \ {⊥}
    send ⟨s, chosen[c], hints⟩ to c
```

and $d.last\_seq[c]$. Note that d never removes a chooser from these tables.

Listings 2 and 3 together give the chooser's implementation, which includes its state, communication predicates and routines, and its four Actions A through D. We separate the chooser's description into two listings for readability; Listing 2 shows the routines, predicates and state used to implement FIFO channels whereas Listing 3 includes the chooser's actions and related state.

A chooser c stores the set of deciders that it knows exists ($c.deciders$), the sequence number of its current choice ($c.seq$), the value of its current choice ($c.me$), hints of choices to avoid according to each decider d ($c.hints[d]$), and the most recent sequence number acknowledged by each decider d ($c.last\_ack[d]$).

The code makes use of a watchdog timer. The timer provides a variable $timeout$ that is true iff the timer is unarmed. The operation TO_arm ensures that the timer is armed (so $timeout$ is false). If TO_arm is not subsequently executed, then $timeout$ eventually becomes true.

A chooser c has the following routines for communication with deciders:

SendTo(s,x,D): Send choice x with sequence number s to all deciders in D.

ResendTo(D): Resend the last message sent to all deciders in D.

ReceiveAck(s,d): Receive an acknowledgment from d on sequence number s.

A chooser c also has three macros to represent some of the re-used code related to channel activities:

HasReceivedAck(d): true iff c has received an acknowledgment from d for its latest choice.

CurrentChoice(s): true iff sequence number s acknowledges c's most recent choice.

OldChoice(s): true iff sequence number s acknowledges an obsolete choice.

These predicates and routines appear along with the associated state in Listing 2. The chooser's actions and related state are shown in Listing 3.

When a chooser needs to select a new value (Action A), it selects one at random, avoiding potentially unavailable values, and sends this to neighboring deciders. It then arms the watchdog timer. When the timer fires (Action B), if the chooser's value has not yet been denied, it resends this selection on any channels necessary. When a chooser receives an acknowledgment from a decider (Action C), it stores the decider's hints if they are up-to-date, and records the sequence number for the acknowledgment. If the message is a rejection, the chooser sets $c.me$ back to undecided so that, via Action A, it will try again. Finally, when a new decider connects to a chooser and the chooser has already sent a proposal to other deciders, it sends its choice to the new decider (Action D). Note that a chooser crashing or recovering has no specific effect in the protocol: a decider only releases the label it has assigned to a chooser c when c asks for a new label. A decider d recovering can cause c to send d its latest choice via Action D.

Listing 2: Chooser Channel Predicates and Routines (Unbounded Channels)

```
int[deciders] last_ack = all[0]

// $\Longleftrightarrow$ c has an ack from d for its latest choice
boolean HasReceivedAck (d):
    last_ack[d] == seq
// $\Longleftrightarrow$ s acknowledges c's most recent choice
boolean CurrentChoice (s):
    s == seq
// $\Longleftrightarrow$ s acknowledges an obsolete choice
boolean OldChoice (s):
    s < seq

SendTo (s,x,D):
    foreach d $\in$ D do
        send $\langle$s,x$\rangle$ to d
ResendTo (D):
    foreach d $\in$ D do
        send $\langle$me,seq$\rangle$ to d
ReceiveAck (s,d):
    last_ack[d] $\leftarrow$ s
```

Listing 3: Chooser Algorithm: Actions and State (Channel Predicates and Routines Separated)

```
set$\langle$Decider$\rangle$ deciders = ...
int seq = 0
Choice me = $\perp$
(set$\langle$Choice$\rangle$)[deciders] hints = all[$\emptyset$]

// when needs to make a choice
A: when me == $\perp$
    choices $\leftarrow$ domain(Choice) \ {$\perp$} \ {hints[d]$_{\forall\, d \in \text{deciders}}$}
    me $\leftarrow$ choose from choices
    seq ++
    SendTo(seq,me,deciders)
    TO_arm
// retransmit last msg sent to deciders yet to acknowledge
B: when timeout $\wedge$ (me $\neq \perp$)
    ResendTo({d $\in$ deciders: $\neg$HasReceivedAck(d)})
    TO_arm
// receive response from d
C: when receive $\langle$s, chosen, hint$\rangle$ from d
    ReceiveAck(s,d)
    if $\neg$OldChoice(s)
        hints[d] $\leftarrow$ hint
    if CurrentChoice(s) $\wedge$ (chosen == $\perp$)
        me $\leftarrow$ $\perp$
// learn of decider d and round is active
D: when detect new decider d $\wedge$ (me $\neq \perp$)
    SendTo(seq,me,{d})
```

This algorithm is not guaranteed to terminate because any pair of choosers can conflict with one another. For example, let choosers c_1 and c_2 both choose the yet-unassigned label x and send it to deciders d_1 and d_2. Decider d_1 may receive c_1's message first and d_2 may receive c_2's message first. Thus, d_1 will reject c_2 and d_2 will reject c_1. This kind of conflict can continue for an unbounded time. However, as long as the domain from which a chooser c selects is large enough, there is a significant probability with each choice that c chooses a label x that is different than any label currently accepted by any decider, and that is different than any label that any other chooser has currently chosen or will choose before c's message with x is received by all deciders. Once this occurs, c's value will be accepted by all deciders. This, in turn, increases the chances that another chooser will have its value chosen. Thus, as the running time tends to infinity, the probability of **Distinctness** holding tends to 1, as we show in Section 5.1.

4.1 Bounding the Channels

This protocol can be modified so that each chooser c limits the number of messages in flight to any given decider. Doing so limits the number of conflicting assignments that might occur in the future from some state: this is useful in computing the expected number of choosers that terminate in a given round (see Section 5.1).

We extend both the basic chooser code as well as its channel code to accommodate channel bounding. In fact, this extension requires only moderate changes to the protocol, as we are able to leverage the variable *seq* that is used to ensure that out-of-date messages are ignored. We add some simple book-keeping to the chooser's channel and some extra logic to the chooser's Action C. We consider the changes to the channel code first.

A chooser c stores the most recent sequence number acknowledged by each decider d ($c.last\_ack[d]$). c also now stores, for each decider d, a set of unacknowledged sequence numbers ($c.sent[d]$), a tuple of the most recent choice and corresponding sequence number sent to d ($c.last\_sent[d]$), and the sequence number of the most recent choice it would have sent to d if it were not limited by available channel space ($c.last\_choice[d]$). The three predicates used for unbounded channels, $HasReceivedAck$, $CurrentChoice$, and $OldChoice$, are modified to make comparisons based on values stored for a particular decider d. That is, they compare a sequence number s to the sequence number of the most recent choice with respect to a decider d ($c.last\_choice[d]$) rather than to a global sequence number *seq*. Choosers also have three new channel predicates:

CanSendTo(d): true iff there is space in the channel from c to d.
SentLatest(d): true iff c has sent its latest choice to d.
RecentAck(s,d): true iff the sequence number s acknowledges c's most recent message to d.

Finally, the *SendTo*, *ResendTo* and *ReceiveAck* routines are updated to include book-keeping and verification, and to send new messages only when there is room in the channel:

SendTo(s,x,D): Send choice x with sequence number s to all deciders in D, keeping a copy for retransmission and bounding the channel.
ResendTo(D): Resend the last message sent (if applicable) to all deciders in D.
ReceiveAck(s,d): Receive an acknowledgment from d on sequence number s, update channel book-keeping variables.

Listing 4: Chooser Channel Predicates and Routines (Bounded Channels)

```
int[deciders] last_ack = all[0]
(set⟨int⟩)[deciders] sent = all[∅]
⟨int,Choice⟩[deciders] last_sent = all[⟨0,⊥⟩]
int[deciders] last_choice = all[0]
int max_in_chan = a non-zero constant

// ⟺ c has an ack from d for its latest choice
boolean HasReceivedAck (d):
    last_ack[d] == last_choice[d]

// ⟺ s acknowledges c's most recent choice for d
boolean CurrentChoice (s,d):
    s == last_choice[d]

// ⟺ s acknowledges an obsolete choice for d
boolean OldChoice (s,d):
    s < last_choice[d]

// ⟺ there is room in the channel to send to d
boolean CanSendTo (d):
    | sent[d] | < max_in_channel

// ⟺ c has sent its most recent choice to d
boolean SentLatest (d):
    last_sent[d][0] == last_choice[d]

// ⟺ s acknowledges c's most recent message to d
boolean RecentAck (s,d):
    s == last_sent[d][0]

SendTo (s,x,D):
    foreach d ∈ D do
        if CanSendTo(d)
            send ⟨s,x⟩ to d
            sent[d] ← sent[d] ∪ {s}
            last_sent[d] ← (s,x)
        last_choice[d] ← s

ResendTo (D):
    foreach d ∈ D do
        if | sent[d] |>0
            send ⟨last_sent[d]⟩ to d

ReceiveAck (s,d):
    sent[d] ← sent[d] \ {i: i≤s}
    last_ack[d] ← s
```

Note that with channel bounding, a chooser maintains the sequence number of the most recent message sent to a decider d ($c.last\_sent[d]$) as well as that of the most recent choice of $c.me$ with respect to d ($c.last\_choice[d]$). A chooser may be temporarily unable to send its current choice to d if the channel between the two is full. This accounts for the subtle difference between the **RecentAck** and **CurrentChoice** predicates. Listing 4 shows the code for the channel-related predicates and routines when channels are bounded.

The chooser's actions change only slightly to accommodate channel-bounding. Actions A and B rely on the now channel-bounding routines *SendTo* and *ResendTo* for sending messages to deciders. This change is encapsulated in the channel code (described above). The chooser's Action C does change; a chooser stores a decider's hints only if the decider is responding to the most recent message *sent to that decider*. Additionally, if an acknowledgment is out-of-date and may have opened space in the channel, the chooser resends its current selection. Listing 5 shows the updated Action C. Modified code is shown in black, while unchanged code is grey.

Listing 5: Chooser Algorithm: Actions and State (Bounded Channels)

```
set⟨Decider⟩ deciders = ...
int seq = 0
Choice me = ⊥
(set⟨Choice⟩)[deciders] hints = all[∅]

// when needs to make a choice
A: when me == ⊥
    choices ← domain(Choice) \ {⊥} \ $\{hints[d]_{\forall d \in deciders}\}$
    me ← choose from choices
    seq ++
    SendTo(seq,me,deciders)
    TO_arm

// retransmit last msg sent to deciders yet to acknowledge
B: when timeout ∧ (me ≠ ⊥)
    ResendTo({d ∈ deciders: ¬HasReceivedAck(d)})
    TO_arm

// receive response from d
C: when receive ⟨s, chosen, hint⟩ from d
    ReceiveAck(s,d)
    if RecentAck(s,d)
        hints[d] ← hint
    if CurrentChoice(s,d) ∧ (chosen == ⊥)
        me ← ⊥
    if OldChoice(s,d) ∧ (me ≠ ⊥)
        SendTo(last_choice[d],me,{d})

// learn of decider d and round is active
D: when detect new decider d ∧ (me ≠ ⊥)
    SendTo(seq,me,{d})
```

5 Analysis of the Decider/Chooser Protocol

In this section, we consider the correctness of DCP, first via proof and then by using model checking software.

5.1 Proof of Correctness of DCP

We prove here that DCP implements LSP. We assume that each channel contains no more than $max\_in\_channel$ messages (Listing 4).

Our proof of correctness uses the following *Eventual Delivery* lemma:

Lemma 1 (Eventual Delivery) *If chooser c sends a message $[seq, me]$ to d, and both c and d remain uncrashed and connected to each other, then eventually d receives a message $[seq', me']$ from c with $seq' \geq seq$, and eventually c receives an acknowledgment from d for a message with a sequence number $seq'' \geq seq$.*

*Proof (**Lemma 1**)* When c sends $[seq, me]$ to d, it will keep sending messages with some sequence number $seq' \geq seq$ to d via Actions A or B until it receives an acknowledgment (via Actions G, C) for $seq'' \geq seq$. ☐

*Proof (**Progress**)* Initially $c.me$ is $\perp$. This variable is set to a non-$\perp$ value only by Action A, and Action A is continuously enabled starting with the initial state. Hence, if c does not remain crashed, $c.me$ will be set to some non-$\perp$ value. ☐

*Proof (**Distinctness**)* A chooser that remains up will execute Action A one or more times. If it executes Action A a final time, we say that the chooser c's choice $c.me$ *stands*: from that point on, $c.me$ does not change. If c's value stands and c remains up, then $c.me \neq \perp$ since, otherwise, Action A is enabled.

We first show that two choosers that share a decider cannot both choose the same label and have their choices stand. That is, if two choosers' values $c_1.me$ and $c_2.me$ stand, then $c_1.me \neq c_2.me$. We then show that with high probability, the choosers will choose distinct values that stand.

(a) It is impossible for two choosers c_1 and c_2, both connected to decider d, to both set $c_1.me = c_2.me = x$ with $x \neq \perp$ and have these values stand. This is because c_1 will send $[s_1, x]$ to d and c_2 will send $[s_2, x]$ to d for some s_1 and s_2. Since both leave me at x, neither sends a message with larger sequence numbers. From Lemma 1, d will eventually deliver both messages, and will reply $\perp$ to at least one of the choosers. Again from Lemma 1 the chooser will receive this acknowledgment and set me to $\perp$.

(b) Consider some point in the execution of the protocol. Let D be the set of deciders and C be the set of choosers. Let C^+ be the subset of choosers that will choose again by

executing Action A — that is, $C \backslash C^{+}$ are the choosers whose choices stand.

If a chooser in C^{+} chooses a value that some decider d has already given to another process, then it may receive $\perp$ from d. There are up to $|D| \times |C|$ distinct values that have already been given by some decider to some chooser. If multiple choosers in C^{+} choose the same value, then some decider d they share may send one of them $\perp$.

If a chooser c in C^{+} chooses a value that is in a message m that was sent by another chooser to a decider d but not yet delivered by d, then d may deliver m before receiving c's choice, and thus d will send $\perp$ to c. There are up to $|D| \times |C| \times max\_in\_channel$ distinct values in channels.

Let $P(q,m,L)$ be the probability that if we take m samples with replacement from a domain of size L, then exactly q of them are distinct. In our case, L corresponds to the label domain, m to the number of choosers still attempting to select values, and q to the number of choosers that choose values that will stand as labels because they are distinct. Let *Choice* be the domain from which choosers choose. Even if all choosers pick distinct values, there are up to

$$|D| \times |C| + |D| \times |C| \times max\_in\_channel$$

values that, if chosen, will result in a chooser receiving $\perp$. Thus, the probability that the choosers in C^{+} all choose values that stand is at least

$$P(|C^{+}|, |C^{+}|, |Choice| - |C| \times |D|(1 + max\_in\_channel))$$

In fact, the probability that some choosers choose values that stand is positive. Thus, with enough choices, C^{+} will continue to decrease with high probability, until it becomes empty. □

5.2 Model Checking DCP

We implemented DCP in Mace [7,12], which is a language for distributed system development. (Our full implementation of DCP can be found at `http://dl.dropbox.com/u/4570403/dcp.tgz`.) The Mace toolkit includes both a model checker [11] that allows one to verify the correctness of the system and a simulator [13] for testing timed behavior. A major benefit of Mace is that Mace code compiles into standard C++ code, which allows one to deploy code that has been model checked.

A few differences between the implementation and the listings of Section 4 bear special mention. A Mace service contains variables and messages, as well as code segments called *transitions* that are executed in reaction to four types of events: timer expiration, message receipt, error indication, and downcalls from applications using the service. As such, Mace cannot constantly test the guards for the actions shown in our listings; instead, we must determine when each guard may become true and evaluate each guard at all necessary points (executing the corresponding action if necessary). A decider's Action G executes upon receipt of any message from a chooser, whereas Action F executes only upon receipt of a message from a chooser that has not yet been encountered. The case for the chooser is a bit more complicated. Action A needs to execute whenever $c.me = \perp$. This can occur initially upon startup of the chooser, upon recovery from a crash (if the value was not set prior to the crash), and as the result of a rejection message in Action C. So, the guard for Action A is evaluated at these three times. The guard for Action B is evaluated when the watchdog timer fires and also upon reset. Action C executes directly as a result of a message receipt from a decider. The guard for Action D is evaluated whenever a chooser receives a message from a decider not currently in $c.deciders$.

Both the Mace model checker and the Mace simulator construct a set of behaviors of the program. Mace knows the sources of nondeterminism (in our case, node failures, UDP packet reordering and loss, and random number generation) and so constructs all behaviors over which it checks for violations of any safety or liveness property. The model checker differs from the simulator in how the sets of behaviors are constructed: the model checker does a breadth first construction while the simulator chooses, at random, a value for each nondeterministic event to construct a behavior. Since the tests cannot be run for an infinite time, each behavior is extended to a maximum depth (set with a runtime parameter).

We used the Mace model checker to check the liveness properties **Progress** and **Distinctness**. We considered three types of topologies, all modifications of a 3-level fat tree.[2] We constructed all three topologies by first creating a 3-level fat tree using k-port switches, with $k = 4$, 6, 8, 10, and 12, and extracting the bottom two levels of nodes. The first topology (*fat tree-based*) consists of this bipartite graph embedded in the lowest two levels of a fat tree. For our *random bipartite* topology, we began with the fat tree-based topology and removed all edges in the graph. We then generated edges between each lower-level node and a randomly chosen set of $\frac{k}{2}$ upper-level nodes. Finally, we also created a *complete bipartite* graph between the nodes within the fat tree-based topology. The complete bipartite graph topology imposes the most restrictions on DCP because all choosers share all deciders: no two choosers can have the same label.

For each topology type, we show that the **Progress** and **Distinctness** properties eventually hold. We also verify the channel bounding aspects of the protocol (see Section 4.1) using safety properties.

---

[2] This topology with arises in the context of ALIAS [19].

6 Performance of the Decider/Chooser Protocol

In this section, we consider the performance of DCP, that is, we explore the time required for an instance of DCP to satisfy **Progress** and **Distinctness**. We begin in Section 6.1 by mathematically analyzing DCP and then we simulate its behavior in Section 6.2.

6.1 Analyzing DCP Performance

Recall that $P(q,m,L)$ expresses the probability that if we take m samples with replacement from a domain of size L, then exactly q of them are distinct. In other words, this value expresses the probability that any given set of choosers will succeed (and therefore exit the competition) during any given round. Therefore, sequences of $P(q,m,L)$ values can form probability distributions for the completion of DCP instances.

$P(q,m,L)$ can be computed as follows. Let $S(m)$ be the set of different sets of positive numbers that sum to m. For example,

$$\begin{aligned} S(6) = \{&\{1,1,1,1,1,1\},\{1,1,1,1,2\},\{1,1,1,3\},\{1,1,4\},\\ &\{1,5\},\{1,1,2,2\},\{1,2,3\},\{2,4\}\\ &\{2,2,2\},\{3,3\},\{6\}\} \end{aligned}$$

We use each element of $S(m)$ to denote a *configuration* of the m choosers. So, $\{1,5\}$ represents a configuration of six choosers in which five choose the same label, and the sixth chooses another label.

Let $C(s)$ be the number of ways the m choosers can be grouped into a configuration s and let $T(s,L)$ be the number of unique ways elements of L can be assigned to configuration s. That is,

$$T(s,L) = |s|! \times \binom{L}{|s|} = \frac{L!}{(L-|s|)!}$$

The probability that m choosers result in configuration s is $C(s) \times T(s,L)/L^m$. For example, let $s = \{1,1,2,2\}$.

$$C(s) = \binom{6}{1} \times \binom{5}{1} \times \frac{\binom{4}{2}}{2!} \times \frac{\binom{2}{2}}{2!} = 45$$

$T(s,10)$ for $s = \{1,1,2,2\}$ is 5,040 and the probability that the choosers are in configuration $\{1,1,2,2\}$ for $L = 10$ is

$$\frac{45 \times 5040}{10^6} = 0.2268$$

Finally, let $S_q(m)$ be the subset of $S(m)$ that contain exactly q values of 1. For example,

$$S_2(6) = \{\{1,1,4\},\{1,1,2,2\}\}$$

Then we have

$$P(q,m,L) = \frac{\sum_{s \in S_q(m)} C(s) * T(s,L)}{L^m}$$

So, $P(2,6,10)$ is

$$\begin{aligned} P(2,6,10) &= \frac{C(\{1,1,2,2\}) \times T(\{1,1,2,2\},10)}{10^6}\\ &+ \frac{C(\{1,1,4\}) \times T(\{1,1,4\},10)}{10^6}\\ &= 0.2268 + 0.0108 = 0.2376 \end{aligned}$$

That is, just under a quarter of the time, if six choosers choose labels from 0 to 9, exactly two will end up with labels distinct from all the other chosen labels. Over 95% of the time that this happens, two other choosers will choose a third label and the remaining two will choose a fourth label, and under 5% of the time the four remaining choosers will choose the same label.

To give an idea of the probability of choosing distinct values, Figure 5 shows a plot for $P(q,32,L)$ for $L = 32$, 64 and 128 (that is, 32 choosers and labels with 5, 6 and 7 bits). With $L = 128$, the most likely value for q is 26, which would leave 6 choosers choosing again. When $L = 32$ (the smallest possible value for L) , the most likely value for q is 12, which leaves 20 choosers choosing again. This shows how decreasing L increases the expected convergence time.

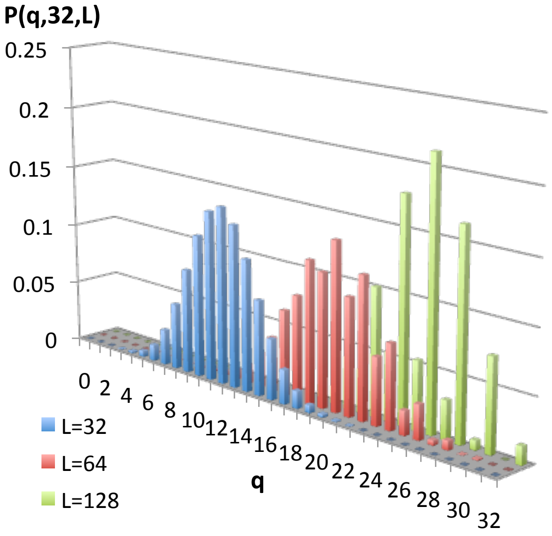

Fig. 5: $P(q,32,L)$ with $L = 32,64,128$

It would be useful to compute an upper bound on the convergence time of DCP, but it has proven difficult to do so: as more choosers choose values that stand, fewer values remain for other choosers, but the number of choosers

Configuration			% Choosers converged vs. Number of Choices									
Topo	|C|	|L|	1	2	3	4	5	6	7	35	55	89
fat tree-based	8	8	94.13	99.88	**100**							
		12	96.38	**100**								
		16	95.50	**100**								
	18	18	94.33	99.94	**100**							
		27	96.50	99.94	**100**							
		36	97.44	**100**								
	32	32	95.38	99.91	**100**							
		48	96.56	**100**								
		64	97.09	99.94	**100**							
	50	50	97.36	**100**								
		75	97.74	**100**								
		100	95.54	99.98	**100**							
	72	72	96.01	99.89	**100**							
		108	97.43	**100**								
		144	98.01	**100**								
random	8	8	88.13	98.75	**100**							
		12	92.88	**100**								
		16	93.88	99.88	**100**							
	18	18	87.89	98.11	99.83	**100**						
		27	92.33	99.17	**100**							
		36	93.94	99.39	99.94	**100**						
	32	32	84.69	98.16	99.88	**100**						
		48	90.16	99.19	99.97	**100**						
		64	92.78	99.53	**100**							
	50	50	83.80	97.02	99.62	99.94	**100**					
		75	89.58	98.92	99.98	**100**						
		100	91.78	99.38	99.98	99.98	**100**					
	72	72	84.19	97.57	99.71	99.96	**100**					
		108	89.40	98.81	99.89	**100**						
		144	92.40	99.29	99.99	**100**						
complete bipartite	8	8	50.00	70.50	81.00	84.88	86.00	87.88	88.88	99.63	**100**	
		12	68.50	91.38	97.75	99.50	99.88	**100**				
		16	75.50	96.25	99.25	**100**						
	18	18	32.17	43.44	50.44	53.56	55.11	56.22	57.61	90.56	98.72	**100**
		27	59.94	84.44	95.17	98.83	99.56	99.83	99.89	**100**		
		36	68.78	91.33	98.28	99.72	**100**					

Fig. 6: Convergence Time of DCP

competing for values decreases. For the purposes of ALIAS, simulation has been sufficient to show that the expected convergence time is short.

6.2 Simulating DCP Performance

After verifying **Progress** and **Distinctness** with the Mace model checker, we used the Mace simulator to determine how quickly DCP converges over the same three types of topologies (fat tree-based, random bipartite and complete bipartite). In addition to the topology type, we varied the number of choosers and deciders[3] ($|C|$) as well as the size of the domain ($|L|$) from which the choices are made. We simulated $|L| = |C|$, which is the smallest domain that allows for a solution with a bipartite graph, $|L| = 1.5|C|$ and $|L| = 2|C|$. For a given number of choosers and deciders, there are 9 possible configurations, corresponding to the three topology types and the three label domain sizes. For each configuration, we simulated 100 different executions (thus giving different values for the nondeterministic events). Figure 6 shows the results of these simulations. Each column gives the percentage of choosers (averaged over 100 executions) that have converged after a given number of choices.

For the first two types of topologies, most choosers converge within 2 choices, and only a few require 3-5 choices before settling on a value. For the complete bipartite graphs, especially when $|L| = |C|$, it takes longer for all choosers to converge because each chooser must choose a distinct value. Even so, in most cases over 90% of the choosers converge with 2 choices and over 99% converge with 4 choices. But, the time for all to decide under such constraints can sometimes be long. For example, in one particular execution for the complete bipartite topology with $|L| = |C| = 18$, the hint messages to a single chooser were repeatedly dropped, and the chooser chose already-taken labels for 89 cycles before converging.

7 DCP in Data Center Labeling

This work arose in the context of automatic label assignment in large-scale data center networks. ALIAS [19] operates over indirect hierarchical topologies [18], in which servers (end hosts) connect to the lowest level of a multi-rooted tree of switches. Such topologies currently underly many data center networks [1,2,5,9,14]. Switches at each level of the hierarchy but the topmost select *coordinates* and these coordinates combine to form hierarchically meaningful labels; a label corresponds to a path from the root of the

---

[3] The number of deciders is equal to the number of choosers.

tree to an end host. In data center networks, a key concern is automatic configuration in the face of a dynamically changing topology, so DCP is well-suited to this environment.

7.1 Distributing the Chooser

Recall that the input to DCP is a bipartite graph of choosers connected to deciders; each chooser and decider resides in a single process. Before we discuss DCP as a solution for coordinate assignment in ALIAS, we first present an extension to the basic protocol, in which a logical chooser can be distributed across multiple nodes. These nodes cooperate to select a single shared label. We will use this extension when we apply DCP within ALIAS's multi-rooted trees in Section 7.2. A full protocol derivation appears in Appendix B.

We begin with the set of nodes that wish to cooperate in order to select a shared label, and introduce a new type of process for these nodes: the *chooser relay*. Each node within the cooperating set functions as a relay, providing a connection from the distributed chooser to one or more deciders. A distributed chooser's set of neighboring deciders consists of the union of all deciders with a direct link to one or more of the chooser's relays. We then introduce another type of process, the *chooser representative*. Each distributed chooser has exactly one representative, which performs all of the functionality of the chooser (Actions A through D of Listing 5), and communicates with deciders via the chooser's relays. This representative can be co-located with one of the relays or it can be a separate node; the only requirement is that it is able to communicate with all of the chooser's relays.

The structure of a distributed chooser with a separately located representative is shown in Figure 7. In the figure, the nodes marked d_1 through d_4 are deciders, and the dotted lines denote the boundaries of the two distributed choosers. Within $\mathit{Chooser}_1$ and $\mathit{Chooser}_2$, rel_1 through rel_5 are relays, and rep_1 and rep_2 are representatives.

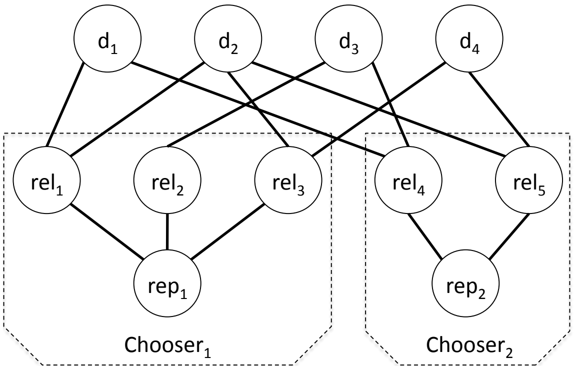

Fig. 7: Distributed Chooser

For a distributed chooser $\mathscr{C}$, we denote with $\mathit{Relays}(\mathscr{C})$ the set of relays in $\mathscr{C}$ and with $\mathit{Repr}(\mathscr{C})$ the process that represents $\mathscr{C}$. Together, the processes in $\mathit{Relays}(\mathscr{C}) \cup \mathit{Repr}(\mathscr{C})$ make up the distributed chooser $\mathscr{C}$. Similarly, for an individual node r, we use $\mathit{Relays}(r)$ and $\mathit{Repr}(r)$ to denote the relays and representative of the chooser in which r participates. In our example of Figure 7, the relays and representatives for the two distributed choosers are as follows:

$$\begin{aligned}\mathit{Relays}(\mathit{Chooser}_1) &= \{\mathit{rel}_1, \mathit{rel}_2, \mathit{rel}_3\}\\ \mathit{Repr}(\mathit{Chooser}_1) &= \mathit{rep}_1\\ \mathit{Relays}(\mathit{rel}_4) &= \{\mathit{rel}_4, \mathit{rel}_5\}\\ \mathit{Repr}(\mathit{rel}_4) &= \mathit{rep}_2\end{aligned}$$

There are some issues to address in implementing a distributed chooser. The first is that of communication between the chooser's representative and its relays. We support this communication with two queues, *Send* and *Receive*:

Send: is a queue of messages stored at each relay r, that implements a virtual channel from $\mathit{Repr}(r)$ to the deciders. $\mathit{Repr}(r)$ appends a message m to this queue by sending a message to $\mathit{Relays}(r)$. When a relay receives this message, it adds m to the end of its own copy of *Send*. $\mathit{Repr}(r)$ never takes an action based on the value of *Send*, and so a relay r need not notify $\mathit{Repr}(r)$ when it removes m from *Send*.

Receive: is a queue of messages stored at $\mathit{Repr}(\mathscr{C})$ that accumulates messages sent to a chooser $\mathscr{C}$ from its deciders. A relay r for chooser $\mathscr{C}$ appends messages to this queue by sending them to $\mathit{Repr}(r)$.

These changes only affect the chooser's actions and channel code slightly; the *SendTo* and *ResendTo* channel functions (Listing 4) append to the *Send* queue rather than sending messages directly to deciders, and a chooser's Action C (Listing 5) is triggered by a non-empty *Receive* queue rather than by direct receipt of a message from a decider.

A second issue has to do with the connections between a distributed chooser and a decider: a chooser may be connected to each of its deciders via various subsets of its relays. Rather than having the representative keep track of which relays are connected to each decider, it can simply send all messages to all of its relays. Each relay then filters out messages destined to deciders that it does not neighbor. While this increases message load, it does not require that a representative keep track of the possibly changing connections between the relays and deciders. Similarly, since a chooser may connect to a decider d via multiple relays, it has the option of selecting only a single such relay for each message sent to d, or it may use any subset of the relays connected to d. This again represents a tradeoff between message load and complexity.

A third issue has to do with data representation at both the chooser and the decider. Since a representative may have multiple paths to a given decider via different relays, it indexes any channel-related variables over both relays and deciders. This is intuitive, as the relays act as virtual channels between a chooser's representative and its deciders. Therefore, channel-related variables should be indexed over the entire channel, relays *and* deciders. Also, recall that a decider indexes its *chosen* and *last_seq* maps over choosers. To support distributed choosers, a decider indexes these maps over the entire chooser, both the relays and the representative.

Finally, changing network conditions may affect connectivity between a chooser's representative and its relays, which can cause the representative to change. Any node that could ever be the representative for a chooser watches the Receive queue and maintains any channel-related state for that chooser. Such a node also executes a modified version of Action C (Listing 5) that properly updates state upon receipt of an acknowledgment so that if it subsequently becomes the chooser's representative, it will have correct acknowledgment and channel capacity information.

7.2 The Decider/Chooser Protocol in ALIAS

Figure 8 shows an example multi-rooted tree of switches. In the figure, hosts have been omitted for space and clarity. Switches are categorized as being at levels L_1 through L_3, from the bottom of the tree upwards, and the S_1 through S_{10} notations indicate switches' unique identifiers.

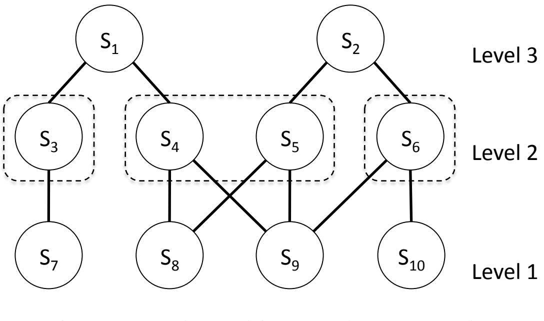

Fig. 8: Sample Multi-Rooted Tree Topology

In ALIAS, an end host h's label is a pair of coordinates c_2c_1, where c_1 is the coordinate of the level L_1 switch l_1 to which h is connected and c_2 is the coordinate of a switch at level L_2 that neighbors l_1.[4] Since there are multiple paths from the root of the tree to an end host h, end hosts in ALIAS have multiple labels. ALIAS forwarding sends data packets to the root of the tree, at which point a packet's destination label specifies a path to the destination. This is based on up*/down* style forwarding, as introduced in Autonet [17].

Since switches forward packets downward based on coordinates within the destination label, it follows that any two children of a given switch should have distinct coordinates; in this way a parent switch can select which child should be the next hop for any given destination label. This maps nicely to a simple application of simultaneous instances of DCP, one per tree level, as we show in Figure 9. Each instance of DCP is used to select coordinates for the instance's choosers. Since there are two levels of switches (L_1 and L_2) that need coordinates, we apply an instance of DCP for each. In the first instance, all L_1 switches act as choosers for their L_1 coordinates and all L_2 switches act as deciders. In the the second instance, all L_2 switches act as choosers for their L_2 coordinates and all L_3 switches are deciders.

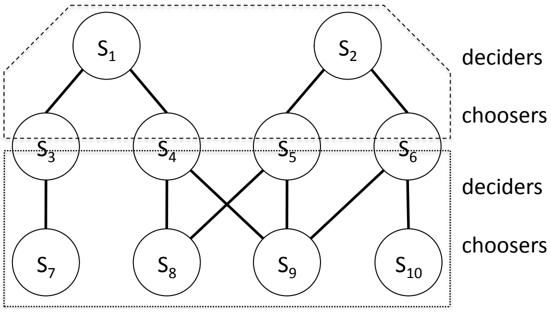

Fig. 9: Simple DCP in ALIAS

The application of DCP to ALIAS L_2 coordinate assignment shown in Figure 9 is simple but not efficient in terms of the number of labels it assigns to each end host. To address this, ALIAS leverages the hierarchical structure of the the topology in order to allow certain sets of switches located near to one another in the hierarchy to share label prefixes. This in turn leads to more compact forwarding state, a desirable property in the data center.

To enable these shared label prefixes, ALIAS introduces the concept of a *hypernode*. In an n-level tree, all switches (other than those at L_n) are partitioned into hypernodes. A hypernode at level L_i is defined as a maximal set of L_i switches that connect to an identical set of L_{i-1} hypernodes below. The base case for this recursive definition has each hypernode at L_1 contain a single switch. For a 3-level tree, the only interesting hypernodes are made up of L_2 switches. Figure 8 shows the sample topology's hypernodes with dotted lines.

Consider a packet with destination label c_2c_1. The coordinate c_1 corresponds to an L_1 switch l_1 that is connected to the packet's destination. Since all L_2 switches in a hypernode connect to the same set of L_1 switches below, an L_3 switch can send the packet to any switch in an L_2 hypernode

---

[4] Top-level switches are not assigned coordinates.

that neighbors l_1. Therefore, the switches in a hypernode can share a single coordinate, as all are equivalent with respect to forwarding reachability. Coordinate sharing among hypernode members reduces the number of labels assigned to an end host and increases the efficiency of ALIAS.

To accommodate shared L_2 coordinates, we apply the distributed chooser version of DCP. Each hypernode corresponds to a single chooser, in which the L_2 member switches are relays. By definition, an L_2 hypernode consists of L_2 switches that connect to the same set of L_1 switches, and so we are guaranteed to have an L_1 switch that can reach all L_2 relays and therefore can act as the chooser's representative. We select between a set of possible representatives via any deterministic function, e.g. the L_1 switch with the smallest MAC address.[5] Figure 10 shows the three distributed choosers for our example topology's L_2 coordinate assignment. These choosers consist of relays $\{s_3\}$ $\{s_4,s_5\}$, and $\{s_6\}$, represented by $\{s_7\}$, $\{s_8\}$, and $\{s_9\}$, respectively.

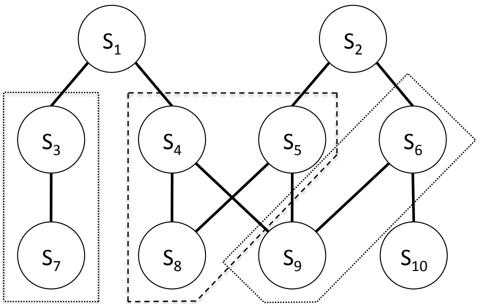

Fig. 10: Assigning Level 2 Coordinates using Distributed Choosers

We have completed a full protocol derivation from Listings 1 and 5 to a complete solution for ALIAS coordinate selection, which we present in Appendix B. A full corresponding Mace implementation is available at `http://dl.dropbox.com/u/4570403/alias.tgz`. We have also built and model checked a second, slightly different implementation of ALIAS[6] with respect to the **Progress** and **Distinctness** properties, and have found through simulation that distributed choosers converge within only a few choices for the networks tested. The full simulation results for our second implementation are reported in a separate article [19].

7.3 Eliminating M-Graphs in ALIAS

The up*/down* forwarding used by ALIAS separates L_1-to-L_n forwarding from L_n-to-L_1 forwarding in an n-level hierarchy. Because of this, a topology that we call an *M-graph* can lead to a forwarding ambiguity. When data forwarding follows an up-down path, two L_1 switches must be no more than $2(n-1)$ hops apart to directly communicate with one another. An M-graph occurs when two L_2 hypernodes hn_1 and hn_2 do not have an L_3 decider in common, and thus may select the same coordinate, but an end host h can communicate with descendants of both hn_1 and hn_2.

An example M-graph is shown in Figure 11. Each switch is marked with a unique identifier (S_1 through S_9) as well as its coordinate if at levels L_1 or L_2. Each host is marked with its unique identifier (h_1 through h_3) and its label (created by concatenating ancestor switches' coordinates). The L_2 hypernodes in the figure are $\{s_3\}$, $\{s_4,s_5\}$, and $\{s_6\}$ and they form distributed choosers represented by $\{s_7\}$, $\{s_8\}$, and $\{s_9\}$, respectively.

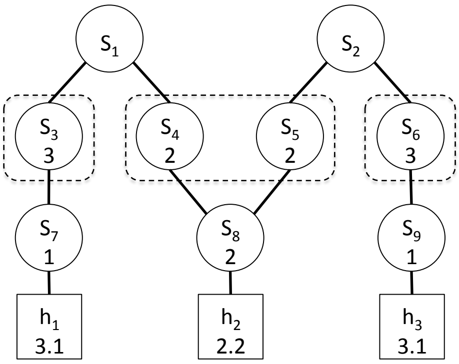

Fig. 11: Example M-Graph

Because data forwarding follows an up-down path, s_7 and s_9 cannot communicate directly with one another. They can, though, both communicate with a third L_1 switch s_8 (and its neighboring host h_2). Since the L_2 hypernodes connected to s_7 and s_9 ($\{s_3\}$ and $\{s_6\}$) do not share a parent they can have the same L_2 coordinate, in this case 3. And, since s_7 and s_9 have no parent in common, they can have the same L_1 coordinate, in this case 1. This is the ambiguity: s_8 can communicate with two different switches, s_7 and s_9, that may legally be assigned the same label.

In practice, this is not a problem because of the randomness of DCP: ambiguous labels are rarely generated. When ALIAS finds such labels, it follows a simple detection-and-recovery approach. If desired, though, we can prevent this ambiguity in two different ways, each involving an application of DCP. First, we can simply add the set of L_1 switches that are 3 hops away from each L_2 hypernode to the set of

---

[5] In general, it is acceptable to use any deterministic function such that the result is identical at all decision points of the function.

[6] Our second implementation does not operate in rounds. Choosers and deciders continuously send messages, ignoring incoming messages that are redundant with respect to already processed information.

deciders for that hypernode's chooser.[7] For example, in Figure 11, s_8 would be a decider for hypernodes $\{s_3\}$ and $\{s_6\}$. This removes the possibility of ambiguity by ensuring that any two hypernodes both reachable from a third L_1 switch have distinct labels. This solution increases implementation complexity slightly, because L_2 relays are not directly connected to all L_1 deciders and so send messages to deciders via tunneling or other similar mechanisms.

Alternatively, one can prevent this ambiguity by assigning coordinates to L_3 switches. In our example, the labels of s_7 and s_9 (and therefore h_1 and h_3) would differ in this new coordinate. To do this, L_3 switches are grouped into hypernodes based on connectivity to L_2 hypernodes. L_3 hypernodes then form distributed choosers, using, for example, common L_1 descendants as representatives. L_1 switches reachable in 2 hops from the L_3 hypernodes are the deciders for this instance of DCP. This approach increases the distance between a chooser's representative and relays. Like the previous solution, this approach leads to indirect connections between relays and deciders. However, unlike the first solution, this method introduces the additional complexity and costs of grouping L_3 switches into hypernodes and assigning L_3 coordinates. For this reason, we would favor the former solution, in which L_1 switches are added to L_2 hypernodes' sets of deciders.

8 The Decider/Chooser Protocol in Wireless Networks

In this section, we describe another example of label assignment based on shared connectivity. This case arises in the context of assigning IP addresses to wireless devices. We offer this example to illustrate a plausible use of DCP outside of the context of data center networking.

A local wireless network, e.g., within a building or a corporation, consists of a set of fixed wireless access points and mobile devices that move around within the network. At any time, a mobile device may be within range of (and may use the same channel as) several access points (APs), but it associates with a single access point at a time. A *handoff* occurs when the device changes its association from one AP to another. If, as a result of handoff, the device needs to acquire a new IP address, then ongoing communication sessions can be disrupted.

There are different ways to avoid this need for a new IP address. For example, the set of access points in a network may utilize a wired distribution system to synchronize with each other, ensuring that an IP address given to a device by AP_1 is permittable for use with AP_2 as well. Or, the APs in a network may communicate with a central server responsible for ensuring IP address uniqueness among all network devices. Managing centralized state or requiring a separate distribution system between APs places a significant additional management burden on the network operator.

A key difficulty of address assignment in this type of network is the dynamism of the network; the set of mobile devices varies over time, as does the set of access points visible to each mobile device. In fact, we learned from speaking with network operators that the issues of changing sets of devices and difficulty with handoff are significant pain points for some types of wireless networks.

This dynamism suggests a solution using the Decider/-Chooser abstraction. In wireless networks, we run an instance of DCP with mobile devices as choosers and access points as deciders, wherein a link between device *md* and AP *ap* indicates that *md* is within range of *ap*, as shown in Figure 12. A mobile device selects an IP address that is acceptable with respect to all APs within range, i.e. all of its deciders. As a device moves throughout the network, its set of deciders change, and if at any time it finds its IP address to be in conflict (as reported by one of its deciders) it reselects. This application of DCP has the benefit of removing the requirement of a central authority or separate wired distribution system between APs, but without the need for IP address reassignment on every handoff.

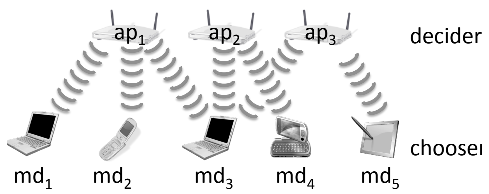

Fig. 12: Multiple AP Example

9 Context and Related Work

The problem we consider here arose in the context of automatic address assignment in large-scale data center networks, specifically, in the design of ALIAS [19], a protocol for automatically assigning hierarchically meaningful labels, or addresses, in such networks.

Our solution uses a Las Vegas type randomized algorithm: the labels that are computed always satisfy the problem specification, but the algorithm is only probabilistically fast. It is also a fully dynamic algorithm [10], in that it makes use of previous solutions to solve the problem more quickly than by recomputing from scratch.

Assigning labels to nodes is not a new problem. For example, in [8] the authors consider the issues of assigning labels to nodes in an anonymous network of unknown size. The quality of an assignment algorithm depends on the size

---

[7] More generally, for an L_i hypernode, we add to the deciders all L_1 switches that are $2n-i-1$ hops from L_1.

of the label domain and the algorithm's efficiency is based on the convergence time and message load. The authors' approach uses a special *source* node (the sole source of asymmetry) to root a spanning tree of the anonymous network, and explores the cost of propagating enough information to label all nodes. We consider networks with significant symmetry: each network can be partitioned into bipartite graphs of processes, even if a process may be made up of multiple nodes. This symmetry and the use of randomization allows us to devise an algorithm in which nodes only communicate with immediate neighbors. This reduces the overall message load relative to that of a network with only a single designated node.

Our solution can also be considered an instance of the *renaming* problem [3,4,6] in which a set of processes, each with a unique name chosen from some large name space, together assign themselves unique names from a smaller name space. The protocol in [6]–which is for a shared memory model–has a similar structure to DCP with a single decider process: our decider has a role similar to a shared atomic snapshot object in their protocol. Their protocol differs in that they sought a deterministic solution; DCP can rename into a smaller name space because it is randomized. Also, LSP differs from the renaming problem: in LSP, two processes can assign themselves the same (shorter) name if they don't share a decider.

Finally, the Label Selection Problem also relates to the graph coloring problem (GCP). In fact, GCP is reducible to LSP. The mapping from GCP to LSP is quite simple; vertices in an instance of GCP, $G = (V,E)$, correspond in a one-to-one mapping to choosers in LSP, and for any pair of vertices in G that are connected by an edge in E we create a decider d and connect each of the corresponding choosers to d. In this way, pairs of vertices that require different colors in GCP correspond to pairs of choosers that require distinct coordinates in LSP. The mapping from LSP to GCP is equally simple. Even though LSP can be mapped to GCP, the LSP structure arises naturally in many protocol problems–like those given in this paper–and the separation of processes into choosers and deciders has helped us to refine DCP for more practical application. However, some techniques for graph coloring could be applied to LSP; for instance one could apply the multi-trials technique introduced by Schneider and Wattenhofer [16] to LSP.

10 Conclusion

This paper considers a version of the network node labeling problem where (1) labels are restricted based on connectivity and (2) the connectivity can change. We call this the Label Selection Problem. We give a Las Vegas style protocol, which we call the Decider/Chooser Protocol, that solves this problem in an efficient manner, and apply this protocol to the problem of automatic label assignment in data center networks. We verify the correctness of DCP via proof and model checking, and explore its performance through analysis and simulation. We find that DCP is quite quick to converge, even with a small label domain, due to the random nature of the protocol. Thus, DCP is particularly well-suited to the practical needs of the data center environment.

References

1. Cisco Data Center Infrastructure 2.5 Design Guide. `http://www.cisco.com/univercd/cc/td/doc/solution/`.
2. M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.
3. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an Asynchronous Environment. *Journal of the ACM*, 37:524–548, July 1990.
4. H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.
5. T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.
6. S. Chaudhuri, M. Herlihy, and M. R. Tuttle. Wait-Free Implementations in Message-Passing Systems. *Theoretical Computer Science*, 220(1):211–245, June 1999.
7. D. Dao, J. Albrecht, C. Killian, and A. Vahdat. Live Debugging of Distributed Systems. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 94–108, Berlin, Heidelberg, 2009. Springer-Verlag.
8. P. Fraigniaud, A. Pelc, D. Peleg, and S. Pérennes. Assigning Labels in Unknown Anonymous Networks (Extended Abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 101–111, New York, NY, USA, 2000. ACM.
9. A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.
10. M. R. Henzinger and V. King. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *Journal of the ACM*, 46(4):502–516, July 1999.
11. C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI '07, pages 18–18, Berkeley, CA, USA, 2007. USENIX Association.
12. C. Killian, J. W. Anderson, R. B. R. Jhala, and A. Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.
13. C. Killian, K. Nagarak, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding Latent Performance Bugs in Systems Implementations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 17–26, New York, NY, USA, 2010. ACM.

14. R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 39–50, New York, NY, USA, 2009. ACM.
15. F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computer Surveys (CSUR)*, 22(4):299–319, Dec. 1990.
16. J. Schneider and R. Wattenhofer. A New Technique for Distributed Symmetry Breaking. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 257–266, New York, NY, USA, 2010. ACM.
17. M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. Autonet: A High-speed, Self-configuring Local Area Network Using Point-to-point Links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318 –1335, October 1991.
18. H. J. Siegel and C. B. Stunkel. Inside Parallel Computers: Trends in Interconnection Networks. *IEEE Computer Science & Engineering*, 3:69–71, September 1996.
19. M. Walraed-Sullivan, R. N. Mysore, M. Tewari, Y. Zhang, K. Marzullo, and A. Vahdat. ALIAS: Scalable, Decentralized Label Assignment for Data Centers. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 6:1–6:14, New York, NY, USA, 2011. ACM.

A The Label Selection Problem with Consensus

In this appendix, we discuss the difficulty of solving LSP with Consensus, beginning with a simple example. Assume the choosers and deciders are connected with a complete bipartite graph. One can implement a Paxos-based state machine in which the choosers implement both the clients of the state machine and the learners of Paxos, and the deciders implement the proposer and acceptors of Paxos, as illustrated in Figure 13. A proposer and an acceptor (e.g. nodes d_2 and d_3 in the figure) can communicate by relaying via a chooser, selected randomly for each message to ensure liveness in the face of crashed choosers. One can implement the state machine so that the client (chooser) that submits the first command is given label 0, the second client is given label 1, etc. Or, one can have each client c choose a random $c.me$ and send it to the state machine; if $c.me$ has been previously requested, then c chooses a label that it has not yet learned has been assigned and tries again. As long as no more than a minority of the deciders remain down (any number of choosers can remain down), this protocol implements the **Progress** and **Distinctness** properties of LSP.

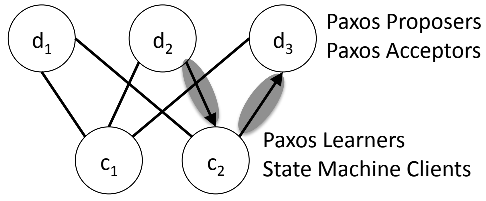

Fig. 13: Simple Consensus Example

If not all choosers have the same set of deciders, then using Consensus becomes messy. The Paxos state machine approach given above can be used by flooding all communication, thereby virtually connecting all processes. This has the drawback of possibly sending excessive messages; the path between any two processes can be as long as the total number of processes. It also unnecessarily restricts the choices of choosers not sharing a decider: all choosers' values will be unique even if they don't share deciders.

Another approach, and one that would not add such unnecessary restrictions to the choices, is to use multiple state machines. Any two choosers that share a decider use a common state machine to agree on unique labels. For example, consider the scenario shown in Figure 14. A valid set of choices is $c_1.me = c_3.me = 0$, and $c_2.me = 1$. One could have two Paxos state machines, one with c_1, c_2, d_1, d_2 and one with c_2, c_3, d_3, d_4. In this approach, client c_2 chooses $c_2.me$ at random and sends it to both state machines. If $c_2.me$ has been previously assigned by either state machine, then it chooses another label and tries again.

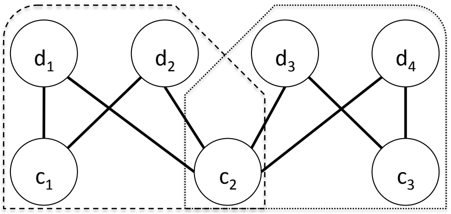

Fig. 14: Complex Consensus Example

This approach has its own set of problems. In this example, if any decider crashes then the solution is not live, because each instance of Paxos can tolerate only a minority of failures; with only two deciders, no permanent crashes can be tolerated. In addition, determining the set of state machines to run is not simple. The set can change as links and switches fail and recover, which adds further complexity.

B From DCP to ALIAS Coordinate Selection

In this appendix we present the full derivation of the ALIAS [19] protocol from the basic version of DCP. We first review the ALIAS environment and details, as well as the basic chooser and decider algorithms. Next, we discuss hypernode calculation, and we refine the chooser to select multiple coordinates simultaneously. Finally, we apply the distributed chooser refinement described in Section 7.1. We present our derivation in the context of a 3-level tree. Though our solution extends to trees of arbitrary depth, we use this limitation for readability.

B.1 ALIAS and DCP Review

Recall that ALIAS switches form an indirect hierarchical topology [18] of n levels, with end hosts connected to switches at the lowest level, L_1. Switches select coordinates that are combined to form topologically meaningful labels; coordinates concatenate along a path from the root of the tree to an end host in order to form a label for that end host. Since there are multiple paths from the root of the tree to any given end host, end hosts have multiple labels.

ALIAS switches are grouped into *hypernodes*: L_i switches that connect to identical sets of L_{i-1} hypernodes form L_i-*hypernodes* that share a single coordinate. Each switch at L_1 is in its own hypernode, and switches at the root of the tree are not grouped into hypernodes as they do not require coordinates. Each L_i switch is a member of exactly one hypernode,[8] and L_i switches may be connected to L_{i+1}

---

[8] The set of L_i hypernodes forms a set of equivalence classes over the L_i switches in a topology.

Listing 6: Decider Algorithm
(Repeated from Listing 1)

```
set⟨Chooser⟩ choosers = ...
Choice[choosers] chosen = all[-1]
int[choosers] last_seq = all[0]

// when connected to new chooser c
F: when new chooser c
    choosers ← choosers ∪ {c}
    chosen[c] ← -1
    last_seq[c] ← 0

// respond to a message from chooser c
G: when receive ⟨s, x⟩ from c
    if s≥last_seq[c]
        last_seq[c] ← s
        if ∃ c' ∈ (choosers \ {c}): chosen[c'] == x
            chosen[c] ← -1
        else
            chosen[c] ← x
    hints ← {chosen[c']$_{\forall c' \in (\text{choosers} \setminus \{c\})}$} \ {-1}
    send ⟨s, chosen[c], hints⟩ to c
```

switches in multiple L_{i+1}-hypernodes. Coordinate sharing within hypernodes serves to ultimately reduce the number of labels per end host in ALIAS. In a 3-level topology, only L_2 switches are grouped into hypernodes; L_1 hypernodes are trivial, with one L_1 switch per hypernode, and L_3 switches are at the root of the hierarchy and do not require coordinate assignments or hypernodes.

Listing 7: Chooser Algorithm: Actions and State
(Bounded Channels, Repeated from Listing 5)

```
set⟨Decider⟩ deciders = ...
int seq = 0
Choice me = -1
(set⟨Choice⟩)[deciders] hints = all[∅]

// when needs to make a choice
A: when me == -1
    choices ← domain(Choice) \ {-1} \ {hints[d]$_{\forall d \in \text{deciders}}$}
    me ← choose from choices
    seq ++
    SendTo(seq,me,deciders)
    TO_arm

// retransmit last msg sent to deciders yet to acknowledge
B: when timeout ∧ (me ≠ -1)
    ResendTo({d ∈ deciders: ¬HasReceivedAck(d)})
    TO_arm

// receive response from d
C: when receive ⟨s, chosen, hint⟩ from d
    ReceiveAck(s,d)
    if RecentAck(s,d)
        hints[d] ← hint
    if CurrentChoice(s,d) ∧ (chosen == -1)
        me ← -1
    if OldChoice(s,d) ∧ (me ≠ -1)
        SendTo(last_choice[d],me,{d})

// learn of decider d and round is active
D: when detect new decider d ∧ (me ≠ -1)
    SendTo(seq,me,{d})
```

We begin our derivation by repeating the basic algorithms for the decider's actions (Listing 1) and the chooser's actions (Listing 5) and channel code (Listing 4), in Listings 6, 7, and 8, respectively. There is one small change to the chooser's channel code: we add routines to clear a chooser's channel corresponding to a particular decider, and to copy channel state from one of chooser's deciders to another. Also, we replace the null coordinate value ⊥ with -1, as this corresponds to the null value of a coordinate in the implementation of ALIAS.

Listing 8: Chooser Channel Predicates and Routines
(Bounded Channels, Repeated from Listing 4)

```
int[deciders] last_ack = all[0]
(set⟨int⟩)[deciders] sent = all[∅]
⟨int,Choice⟩[deciders] last_sent = all[⟨0,-1⟩]
int[deciders] last_choice = all[0]
int max_in_chan = a non-zero constant

// ⟺ c has an ack from d for its latest choice
boolean HasReceivedAck (d):
    last_ack[d] == last_choice[d]

// ⟺ s acknowledges c's most recent choice for d
boolean CurrentChoice (s,d):
    s == last_choice[d]

// ⟺ s acknowledges an obsolete choice for d
boolean OldChoice (s,d):
    s < last_choice[d]

// ⟺ there is room in the channel to send to d
boolean CanSendTo (d):
    | sent[d] | < max_in_channel

// ⟺ c has sent its most recent choice to d
boolean SentLatest (d):
    last_sent[d][0] == last_choice[d]

// ⟺ s acknowledges c's most recent message to d
boolean RecentAck (s,d):
    s == last_sent[d][0]

SendTo (s,x,D):
    foreach d ∈ D do
        if CanSendTo(d)
            send ⟨s,x⟩ to d
            sent[d] ← sent[d] ∪ {s}
            last_sent[d] ← (s,x)
        last_choice[d] ← s

ResendTo (D):
    foreach d ∈ D do
        if | sent[d] |>0
            send ⟨last_sent[d]⟩ to d

ReceiveAck (s,d):
    sent[d] ← sent[d] \ {i: i≤s}
    last_ack[d] ← s

ClearChannel (d):
    last_ack[d]←0
    sent[d].clear()
    last_sent[d] ← (0,-1)
    last_choice[d]←0

CopyChannel (d,ref):
    last_choice[d]←last_choice[ref]
```

We first consider the computation of hypernodes before continuing with the remainder of the derivation in Section B.3.

B.2 Computing Hypernodes

Prior to assigning coordinates, ALIAS hypernodes need to be identified. We select a *representative L_1 switch* for each L_i hypernode via a deterministic function, e.g. the L_1 switch with the smallest UID (in our implementation, MAC address) among those reachable via $(i-1)$ downward hops from switches in the hypernode. This L_1 switch functions as a distributed chooser's representative (Section 7).

Listings 9 and 10 show the actions executed by L_2 switches and L_1 switches, respectively, for computing hypernodes and representative L_1 switches. In Action P, each time an L_2 switch's set of neighboring L_1 switches changes, it sends this set of neighboring L_1 switches to all of its L_1 neighbors.[9] An L_1 switch stores this set (Action Q) and computes the sending L_2 switch's hypernode. Regardless of whether they represent any hypernodes, all L_1 switches perform computation to determine the set of L_2 hypernodes to which they are connected. An L_1 switch runs nearly identical code (omitted for space) when it detects the disconnection of an L_2 switch. There is also logic to ensure that messages are eventually delivered, and that they are delivered in order. This code is also omitted from the listings for brevity.

Listing 9: Hypernode Computation: L_2 Switches

```
set⟨Switch⟩ $L_1$s = ...          // corresponds to choosers of Listing 6
set⟨Switch⟩ $L_3$s = ...

// when $L_1$ neighbors change
P: when detect change in $L_1$s
    foreach n ∈ {$L_1$s ∪ $L_3$s} do
        send ⟨$L_1$s⟩ to n
```

Listing 10: Hypernode Computation: L_1 Switches

```
set⟨Switch⟩ $L_2$s = ...          // corresponds to deciders of Listing 12
(set⟨Switch⟩)[$L_2$s] $L_1$_sets = all[∅]
(set⟨Switch⟩)[$L_2$s] HN = all[∅]        // corresponds to HN of Listing 12

// on notification from $L_2$ switch
Q: when receive ⟨$L_1$s⟩ from s ∈ $L_2$s
    $L_1$_sets[s] ← $L_1$s
    HN[s] ← {s}
    foreach n ∈ {$L_2$s \ {s}} do
        if $L_1$_sets[n] == $L_1$_sets[s]
            HN[s] ← HN[s] ∪ {n}
    foreach n ∈ HN[s] do
        HN[n] ← HN[s]
```

B.3 L_1-coordinate Assignment: Basic DCP

In this section, we discuss the assignment of L_1 coordinates to ALIAS switches using DCP. We consider two options for L_1 coordinate selection and discuss the tradeoffs associated with each.

Recall that to assign L_1 coordinates in ALIAS, we can simply apply DCP, with L_1 switches as choosers and L_2 switches as deciders. Note that a single L_1 switch may be participating as a chooser with respect to several different sets of shared deciders. That is, chooser c_1 may share deciders d_1 and d_2 with chooser c_2 and deciders d_3 and d_4 with chooser c_3. In fact, these sets of shared deciders correspond exactly to the L_2 hypernodes in the topology.

There are two options for L_1 coordinate selection in ALIAS. Both satisfy the **Distinctness** property of LSP amongst L_1 switches:

1. **Single L_1 Coordinate**: On one hand, we can assign a single L_1 coordinate c_1 to each L_1 switch. In this case, the set of labels for an L_1 switch s_1 will be of the form $\{(c_{2\_1},c_1),\ldots,(c_{2\_m},c_1)\}$ where c_{2_1} through c_{2_m} are the L_2 coordinates of each of the m hypernodes to which l_1 is connected.
2. **L_1 Coordinate Per L_2 Hypernode**: Another option is to assign to l_1 multiple L_1 coordinates, one per neighboring L_2 hypernode. Here, l_1's label set will be of the form $\{(c_{2\_1},c_{1\_1}),\ldots,(c_{2\_m},c_{1\_m})\}$, and l_1 will have an L_1 coordinate corresponding to each neighboring L_2 hypernode (and therefore each L_2 coordinate c_{2_i}).

There are tradeoffs between these two options. With option (1), we have a simpler protocol; l_1 only needs to select and keep track of one coordinate. However, this scheme may unnecessarily restrict l_1's coordinate choices, forcing the coordinate domain to be larger than necessary. This is because l_1 may compete with every other L_1 switch in the topology for its coordinate, even if it shares a different set of L_2 deciders with each other L_1 switch. Additionally, this scheme may result in extra communication on topology changes. A topology change that introduces a connection between an L_1 switch l_1 and L_2 switch l_2 forces l_1 and all of its neighboring L_2 switches to rerun DCP. This could potentially involve all L_2 switches in the topology, even those outside of l_2's hypernode. Option (2) provides the complement of these tradeoffs; it is more complex to implement, but reduces the required size of the coordinate domain to the largest set of L_1 switches all connected to an L_2 hypernode. Additionally, after a topology change, an L_1 switch only needs to communicate with the L_2 switches in a single hypernode.

We illustrate these tradeoffs in Figure 15. Suppose the dotted link is initially not present. In this case, regardless of the option used, each L_1 switch has only a single coordinate, as each only connects to one L_2 hypernode. Because s_5 and s_6 do not share deciders, they are free to have the same coordinate, in this case 7. Initially, s_5 has only a single label in its set, $\{3.7\}$. Suppose that the dotted link now appears, causing s_5 to share a decider with s_6. Under option (1), s_5 will have to select a new coordinate, and will have to communicate with all neighboring L_2 switches (in this example, all L_2 switches in the topology) to discover that it cannot select 1 or 7. If it selects $x \neq 1,7$, its new label set becomes $\{3.x, 4.x\}$, and the coordinate domain must include at least 3 choices. On the other hand, with option (2), s_5 only reselects its coordinate with respect to hypernode $\{s_3\}$, and can select a second coordinate that is anything other than 7. s_5 only communicates with s_3 to accomplish this, and its new label set is $\{3.7, 4.x\}$, with $x \neq 7$, giving an overall coordinate domain size of 2.

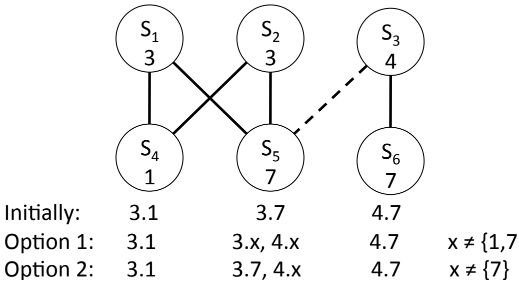

Fig. 15: Two Options for L_1-Coordinate Selection

---

[9] It also sends this set to neighboring L_3 switches to facilitate its own hypernode's coordinate assignment, as explained in Section B.4.

We can implement the first option by simply running a single instance of DCP: L_1 switches take the role of choosers and L_2 switches are deciders. This approach uses the exact algorithms of Listings 6 through 8. However, because of the tradeoffs discussed above, ALIAS adopts the second option for L_1-coordinate selection; it assigns to each L_1 switch l_1, a set of coordinates, one for each of l_1's neighboring L_2 hypernodes. To implement this, we could run multiple simultaneous instances of DCP at each L_1 switch l_1, one instance for each neighboring L_2 hypernode, in separate processes on l_1. However, this can be costly in terms of performance. Additionally, hypernode membership changes may cause complicated interactions between these DCP instances. Instead, we modify the chooser process to keep track of multiple coordinates at once. We perform this refinement in two steps.

In the first step, we introduce the concept of per-hypernode coordinates into the chooser's actions and state. This is shown in Listing 11. Rather than storing just the set of neighboring deciders (*c.deciders* of Listing 7), a chooser stores the set of neighboring hypernodes in *c.HNs* and a map of hypernodes to their member deciders in *c.deciders*. The chooser indexes *c.me* over its set of neighboring hypernodes, and so all instances of *c.me* from Listing 7 are replaced with $c.me[h]$ in Listing 11. Note that it is not necessary to index *c.seq* over hypernodes, because the only requirement of *c.seq* is that it increase with each choice; it need not increase by exactly 1.

Listing 11: Chooser Algorithm: Actions and State (Multi-Hypernode Refinement 1)

```
set⟨HN⟩ HNs
(set⟨Switch⟩)[HNs] deciders =...
int seq = 0
Choice[HNs] me = all[-1]
(set⟨Choice⟩)[deciders] hints = all[∅]

// when needs to make a choice
A: when $\exists$ h $\in$ HNs: me[h] == -1
    choices $\leftarrow$ domain(Choice) \ {-1} \ $\{\text{hints[d]}_{\forall\, \text{d} \in \text{deciders[h]}}\}$
    me[h] $\leftarrow$ choose from choices
    seq ++
    SendTo(seq,me[h],deciders[h])
    TO_arm

// retransmit last msg sent to deciders yet to acknowledge
B: when timeout
    dests $\leftarrow$ $\{\text{deciders[h]}_{\forall\, \text{h} \in \text{HNs: (me[h]} \neq \text{-1)} \wedge (\neg\text{HasReceivedAck(h))}}\}$
    ResendTo(dests)
    TO_arm

// receive response from d
C: when receive ⟨s, chosen, hint⟩ from d
    choose h $\in$ HNs: d $\in$ deciders[h]
    ReceiveAck(s,d)
    if RecentAck(s,d)
        hints[d] $\leftarrow$ hint
    if CurrentChoice(s,d) $\wedge$ (chosen == -1)
        me[h] $\leftarrow$ -1
    if OldChoice(s,d) $\wedge$ (me[h] $\neq$ -1)
        SendTo(last_choice[d],me[h],{d})

// decider d joins HN h and round is active
D: when $\exists$ d $\in$ deciders, h $\in$ HNs: (d joins deciders[h]) $\wedge$ (me[h] $\neq$ -1)
    choose d' $\in$ deciders[h]: d' $\neq$ d
    hints[d] $\leftarrow$ ∅
    ClearChannel(d)
    CopyChannel(d,d')
    SendTo(seq,me[h],{d})
```

The guards and pseudocode for Actions A, B, and D change to incorporate the notion of a hypernode; when a chooser needs to make a choice *for a particular hypernode*, Action A executes, Action B resends to only those *hypernodes* that require retransmission[10], and Action C is updated to determine the hypernode to which the sending decider belongs. When a chooser learns that a new decider has joined a hypernode, Action D executes and uses channel routines *CopyChannel* and *ClearChannel* to enable a new hypernode member to "catch up" with the other members. Here, we define *joins* as the moment when d moves from $deciders[h_1]$ to $deciders[h_2]$, with $h_1 \neq h_2$ and $|h_2| \geq 2$.

The refinement above is intuitive, but not directly implementable, as we have no concrete representation for a hypernode. We address this with our second step in Listing 12, by introducing the following representation: To index a variable over a hypernode, we index it over all individual member switches of the hypernode. To read a value of a hypernode (e.g. $c.me[h]$), we read the corresponding value from any decider in the hypernode, and to write a value to a hypernode, we write to all members of the hypernode.

To keep track of neighboring deciders and hypernodes, a chooser c stores the set of neighboring deciders (*c.deciders*) and a map of each decider d to the set of deciders in d's hypernode (*c.HN*). While *c.me* was indexed over hypernodes in Listing 11, it is indexed over all deciders in Listing 12. When the value of *c.me* is to be written for a particular hypernode, it is written for all deciders in that hypernode, and when it is read, it is read from a single member of the hypernode. The guard for Action A, the set of deciders to receive resent messages in Action B, and the operations in Action D are all updated to accommodate these changes. In Action D, we define"joins" as the moment at which d moves from $HN[d_1]$ to $HN[d_2]$, with $d_1 \neq d_2$ and $|HN[d_2]| \geq 2$.

Note that hypernode computation runs simultaneously with this instance of DCP, with L_1s of Listing 9, L_2s of Listing 10, and *HN* of Listing 10 corresponding to *choosers* (Listing 6), and *deciders* and *HN* (Listing 12) respectively. We transition to these variable names in our next refinement. Each L_2 switch belongs to exactly one hypernode and therefore participates in exactly one instance of DCP. So, the code for the decider does not change from that of Listing 6 for this refinement. The chooser's channel-related code also remains as in Listing 8.

B.4 L_2-coordinate Assignment

We next discuss the assignment of L_2-coordinates to L_2 hypernodes. We use the extension of DCP introduced in Section 7.1 to allow each L_2 hypernode to function as a distributed chooser, with neighboring L_3 switches as deciders. However, before giving the refinement for this extension, we first consider the necessity of a distributed chooser for L_2 coordinate selection.

A tempting approach is to use one instance of DCP in which L_3 switches are deciders and a single L_2 switch from each hypernode is a chooser. However, this does not work. For example, refer to the network in Figure 8 (Section 7). There are three hypernodes: $\{s_3\}$, $\{s_4,s_5\}$, and $\{s_6\}$. The L_2-coordinate shared by s_4 and s_5 must be distinct from that of s_3 and that of s_6. Thus, whatever implements the chooser for the hypernode $\{s_4,s_5\}$ needs to communicate with the deciders at s_1 and at s_2. Neither s_4 nor s_5 is connected to both deciders, and so s_4 and s_5 must together implement a chooser for their hypernode.

Given that we need the cooperation of all L_2 switches in a hypernode, we apply the extension of DCP introduced in Section 7.1 for L_2 coordinate selection. Recall that this extension distributes a chooser $\mathscr{C}$ into a set, $Relays(\mathscr{C})$, of processes that all share a common coordinate as well as a single process, $Repr(\mathscr{C})$, that performs the choosers

---

[10] The astute reader may notice that the channel predicate *HasReceivedAck* operates over a hypernode rather than a decider. This temporary inconsistency will be resolved in our next refinement.

Listing 12: Chooser Algorithm: Actions and State (Multi-Hypernode Refinement 2)

```
set⟨Switch⟩ deciders = ...                  // corresponds to $L_2$s of Listing 9
(set⟨Switch⟩)[deciders] HN = ...           // corresponds to HN of Listing 9
int seq = 0
Choice[deciders] me = all[-1]
(set⟨Choice⟩)[deciders] hints = all[∅]

// when needs to make a choice
A: when ∃ d ∈ deciders: me[d] == -1
    choices ← domain(Choice) \ {-1} \ {hints[d']$_{\forall d' \in HN[d]}$}
    ME ← choose from choices
    foreach d' ∈ HN[d] do
        me[d'] ← ME
    seq ++
    SendTo(seq,ME,HN[d])
    TO_arm

// retransmit last msg sent to deciders yet to acknowledge
B: when timeout
    dests ← {d ∈ deciders: (me[d] ≠ -1) ∧ (¬HasReceivedAck(d))}
    ResendTo(dests)
    TO_arm

// receive response from d
C: when receive ⟨s, chosen, hint⟩ from d
    ReceiveAck(s,d)
    if RecentAck(s,d)
        hints[d] ← hint
    if CurrentChoice(s,d) ∧ (chosen == -1)
        foreach d' ∈ HN[d] do
            me[d'] ← -1
    if OldChoice(s,d) ∧ (me[d] ≠ -1)
        SendTo(last_choice[d],me[d],{d})

// decider d joins d''s HN and round is active
D: when ∃ d, d' ∈ deciders: (d joins HN[d']) ∧ (me[d'] ≠ -1)
    me[d] ← me[d']
    hints[d] ← ∅
    ClearChannel(d)
    CopyChannel(d,d')
    SendTo(seq,me[d],{d})
```

Listing 13: Chooser Algorithm: Actions and State (Distributed Chooser, Representative L_1 Switches)

```
set⟨Switch⟩ $L_2$relays
(set⟨Switch⟩)[$L_2$relays] HN = ...
(set⟨Switch⟩)[$L_2$relays] deciders = ...
int seq = 0
Choice[$L_2$relays] me = all[-1]
((set⟨Choice⟩)[$L_2$relays] hints = all[∅]

// when needs to make a choice
A: when ∃ $l_2$ ∈ $L_2$relays: me[$l_2$] == -1
    choices ← domain(Choice) \ {-1} \ {hints[$l_2$']$_{\forall l_2' \in HN[l_2]}$}
    ME ← choose from choices
    foreach $l_2$' ∈ HN[$l_2$] do
        me[$l_2$'] ← ME
    seq ++
    dests ← {d ∈ deciders[$l_2$']$_{\forall l_2' \in HN[l_2]}$}
    SendTo(seq,ME,$l_2$,dests)
    TO_arm

// retransmit last message sent to deciders yet to acknowledge
B: when timeout
    foreach $l_2$ ∈ $L_2$relays: me[$l_2$] ≠ -1 do
        dests ← {d ∈ deciders[$l_2$]: ¬HasReceivedAck(d,$l_2$)}
        ResendTo(dests,$l_2$)
    TO_arm

// receive response from d
C: when ¬Receive.empty()
    [s,chosen,hint,rep_$l_1$,d,$l_2$] ← Receive.removeHead()
    ReceiveAck(s,d,$l_2$)
    if RecentAck(s,d,$l_2$)
        hints[$l_2$] ← hint
    if CurrentChoice(s,d,$l_2$) ∧ (chosen == -1)
        foreach $l_2$' ∈ HN[$l_2$] do
            me[$l_2$'] ← -1
    if OldChoice(s,d,$l_2$) ∧ (me[$l_2$] ≠ -1)
        SendTo(last_choice[d][$l_2$],me[$l_2$],$l_2$,{d})
```

Listing 14: Chooser Algorithm: Actions and State (Distributed Chooser, L_2 Relay)

```
Switch myID

// when data to send
S: when ¬Send.empty()
    [s,x,hn,rep_$l_1$,d] = Send.removeHead()
    send ⟨s,x,hn,rep_$l_1$⟩ to d

// when data to receive
R: when receive ⟨s,chosen,hint,rep_$l_1$⟩ from d
    Receive.append([s,chosen,hint,rep_$l_1$,d,myID])
```

actions. Listings 13 and 14 contain the chooser's actions and state for $Repr(\mathscr{C})$ and $Relays(\mathscr{C})$, respectively.

As shown in Listing 13 a chooser's representative maintains the set of L_2 switches to which it connects ($c.L_2relays$), the hypernode membership of each neighboring L_2 switch ($c.HN$), and the L_3 deciders to which each neighboring L_2 switch connects ($c.deciders$). Since it will compute a value of $c.me$ to be shared by an entire hypernode, a representative needs to index $c.me$ over the set of neighboring hypernodes (in case it represents multiple hypernodes). As in our previous refinement, we index over hypernodes by writing a value for a hypernode to all of its L_2 members and by reading a hypernode's value via any of its L_2 members. Therefore, $c.me$ is indexed over the representative's neighboring L_2 switches. The $c.hints$ variable is index similarly.

Action A is triggered by a hypernode with a null value for $c.me$ (indicated by an L_2 switch with a null value). The representative collects all hints for this hypernode, selects a new choice for the hypernode, and writes this choice to all of the hypernode's L_2 members. As in previous version of the protocol, it then updates its sequence number, determines the deciders that neighbor this hypernode, and sends its choice to the deciders via the appropriate relays.[11] Action B differs slightly from previous version of the protocol, in that it checks for whether a hypernode has made a choice in a for loop rather than in the Action's guard. This is so the chooser can resend on behalf of all necessary hypernodes in one execution of Action B, rather than only resending for a single hypernode when the timer fires. Action C is triggered by a non-empty *Receive* queue rather than by direct receipt of a

---

[11] The representative includes the L_2 switch that triggered this action as an argument for the *SendTo* channel routine, so that the routine can determine the appropriate set of relays for the message.

message from a decider. The representative does not run its own copy of Action *D*, rather all L_1 switches run Action *D* as discussed below.

We next consider the L_2 relays of the distributed chooser, as shown in Listing 14. This listing introduces the two chooser Actions *S* and *R* that partially implement the *Send* and *Receive* queues between the chooser's relays and representative. When a representative sends its choice to a decider, it includes the sequence number, the choice itself, the current hypernode's members for which it is choosing, its own identity, and the decider for which the message is intended. The third and fourth arguments are new in this refinement and are used at the decider for book-keeping. In Action *S*, an L_2 switch passes the first four parameters to the appropriate decider. Similarly, when a decider responds to a representative's choice, it includes the sequence number, the choice (null if the message is a rejection), a set of hints, and the representative L_1 switch for which the message is intended. An L_2 relay adds the decider's and its own identities to this information and enqueues it on the *Receive* queue for retrieval by the representative via Action *C*.

Recall from Section 7 that all L_1 switches, including non-representatives, execute a version of Action *D*. This is shown in in Listing 15. Action *D* captures situations in which an L_1 switch l_1 newly represents an L_2 relay l_2, either because l_1 has just become a chooser $\mathscr{C}$'s representative or because l_2 switch has just joined *Relays*($\mathscr{C}$). Via Action *D*, the representative resets and copies the associated state, and then resends choices to deciders (via relays) as necessary. Non-representative L_1 switches also maintain and read *Receive* queues for neighboring hypernodes. This is so they have current information on the capacity left in all channels should they become a representative at some point in the future. This is captured via Action C', Listing 15.

Listing 15: Chooser Algorithm: Actions and State (Distributed Chooser, All Neighboring L_1 Switches)

```
// receive response from d
C': when ¬Receive.empty()
    [s,chosen,hint,$l_1$rep,d,$l_2$] ← Receive.removeHead()
    if ¬(AmRep$L_1$($l_2$))
        ReceiveAck(s,d,$l_2$)

// when AmRep$L_1$($l_2$) changes or $l_2$'s HN changes
D: when ∃ $l_2$ ∈ $L_2$relays: AmRep$L_1$($l_2$) becomes true ∨
            ∃ $l_2$, $l_2$' ∈ $L_2$relays: ($l_2$ joins HN[$l_2$']) ∧ (me[$l_2$'] ≠ -1) ∧
                                (AmRep($l_2$'))
    me[$l_2$] ← -1
    hints[$l_2$] ← ∅
    ClearChannel($l_2$)
    if ∃ $l_2$' ∈ $L_2$relays: ($l_2$ joins HN[$l_2$']) ∧ (me[$l_2$'] ≠ -1) ∧
                    (AmRep($l_2$'))
        CopyChannel($l_2$,$l_2$')
        seq++
        dests ← {d ∈ deciders[$l_2$']$_{\forall l_2' \in HN[l_2]}$}
        SendTo(seq,me[$l_2$],$l_2$,dests)
```

The remainder of the changes to a chooser are in its channel routines and predicates, as shown in Listing 16. Since a relay provides a virtual channel to a decider from a representative, the representative indexes all channel variables over the entire virtual channel, decider and relay. This affects all channel-related variables (*sent*, *last_sent*, *last_ack*, and *last_choice*) and the channel-bounding predicates.

The channel code houses the new *Send* and *Receive* queues, and the *SendTo* and *ResendTo* routines append to the *Send* queue rather than sending a message directly to a decider as in previous versions of the protocol. Note that the *SendTo* and *ResendTo* routines enqueue a message intended for a decider *d* onto the *Send* queue of every L_2

Listing 16: Chooser Channel Predicates and Routines (Bounded Channels, Distributed Chooser)

```
int[deciders][$L_2$relays] last_ack = all[0]
(set⟨int⟩)[deciders][$L_2$relays] sent = all[∅]
⟨int,Choice⟩[deciders][$L_2$relays] last_sent = all [⟨0,-1⟩]
int[deciders][$L_2$relays] last_choice = all[0]
int max_in_chan = a non-zero constant

queue[$L_2$relays] Send
queue[$L_2$relays] Receive

// ⟺ c has an ack from d via $l_2$ for its latest choice
boolean HasReceivedAck (d,$l_2$):
    last_ack[d][$l_2$] == last_choice[d][$l_2$]

// ⟺ s acknowledges c's most recent choice to d via $l_2$
boolean CurrentChoice (s,d,$l_2$):
    s == last_choice[d][$l_2$]

// ⟺ s acknowledges an obsolete choice sent to d via $l_2$
boolean OldChoice (s,d,$l_2$):
    s < last_choice[d][$l_2$]

// ⟺ there is room in the channel to send to d via $l_2$
boolean CanSendTo (d,$l_2$):
    | sent[d][$l_2$] | < max_in_channel

// ⟺ c has sent its most recent choice to d via $l_2$
boolean SentLatest (d,$l_2$):
    last_sent[d][$l_2$][0] == last_choice[d][$l_2$]

// ⟺ s acknowledges c's most recent message to d via $l_2$
boolean RecentAck (s,d,$l_2$):
    s == last_sent[d][$l_2$][0]

SendTo (s,x,$l_2$,D):
    foreach d ∈ D do
        if CanSendTo(d,$l_2$)
            foreach $l_2$' ∈ HN[$l_2$]: d ∈ deciders[$l_2$'] do
                Send[$l_2$'].append([s,x, HN[$l_2$],myID,d])
            foreach $l_2$' ∈ HN[$l_2$] do
                sent[d][$l_2$'] ← sent[d][$l_2$'] ∪ {s}
                last_sent[d][$l_2$'] ← (s,x)
        foreach $l_2$' ∈ HN[$l_2$] do
            last_choice[d][$l_2$'] ← s

ResendTo (D,$l_2$):
    foreach d ∈ D do
        if |sent[d][$l_2$]|>0
            foreach $l_2$' ∈ HN[$l_2$]: d ∈ deciders[$l_2$'] do
                Send[$l_2$'].append([last_sent[d][$l_2$'],HN[$l_2$],myID,d])

ReceiveAck (s,d,$l_2$):
    foreach $l_2$' ∈ HN[$l_2$] do
        sent[d][$l_2$'] ← sent[d][$l_2$'] \ {i: i≤s}
        last_ack[d][$l_2$'] ← s

ClearChannel ($l_2$):
    foreach d ∈ deciders[$l_2$] do
        last_ack[d][$l_2$] ← 0
        last_choice[d][$l_2$] ← 0
    foreach $l_2$' ∈ $L_2$relays, d ∈ deciders do
        connects_to_d ← {$l_2$" ∈ HN[$l_2$']: d ∈ deciders[$l_2$"]}
        if connects_to_d == ∅
            last_sent.erase(d,$l_2$')
            last_choice.erase(d,$l_2$')
            last_ack.erase(d,$l_2$')
            sent.erase(d,$l_2$')

CopyChannel ($l_2$,ref,D):
    foreach d ∈ D do
        last_choice[d][$l_2$] ← last_choice[d][ref]
```

switch that reaches d. As discussed in Section 7, a distributed chooser's representative has the option to send a message to a decider d via:

1. every L_2 switch that it neighbors, letting the L_2 switches filter unroutable messages
2. all (or a subset) of its neighboring L_2 switches that reach d, possibly sending the choice to d via multiple relays
3. a subset of its neighboring L_2 switches that reach d, possibly sending the choice to d via multiple relays
4. only one of its neighboringL_2 switch that reaches d.

These options have tradeoffs between synchronization complexity and message load; we favor option (2) as a middle ground.

Finally, the *ClearChannel* function becomes more complicated, as a result of the fact that we represent a hypernode with its constituent L_2 members. Because of this representation, the channel bounding variables may include entries for decider-L_2 switch pairs (d,l_2) for which l_2 is not connected to d, but there is some l_2' in l_2's hypernode that is connected to d. If l_2' leaves the hypernode containing l_2, then any (d,l_2) values need to be removed.

For L_2-coordinate assignment, the decider becomes more complex as well. A decider keeps a record of all L_2 and L_1 switches it has seen ($d.L_2relays$ and $d.L_1reps$). It indexes the choosers that it has seen over $d.L_2relays$ and $d.L_1reps$, representing a chooser via its constituent L_2 members (*d.chooser*). Finally, the decider indexes its choice variables (*chosen*, and *last_seq*) over entire choosers, L_2 relays and L_1 representatives. This is necessary because the representative switch for a hypernode can change. Thus, deciders may maintain duplicate information for a hypernode, namely information obtained from two different switches claiming to represent that hypernode. Recall from Section B.2 that an L_2 switch sends its current set of neighboring L_1 switches to L_3 switches when this set changes. As such, a decider d always knows the most recent set of L_1 switches to which a neighboring L_2 is connected, and d can compute the current representative switch for the hypernode and select the appropriate value of *d.chosen* to pass to an overlying communication protocol. Deciders employ a similar representation for hypernodes as do choosers; they simply index over hypernodes by indexing over the hypernodes' member switches (as shown in Action G).

A decider may be connected to a chooser via multiple L_2 switches, and thus needs to make a decision on whether to accept a value received via an L_2 switch based on the hypernode of the L_2 switch. This adds a small amount of complexity to the decider's Action G; A decider compares a requested value x to those held by L_2 switches in all other hypernodes, regardless of the representative switches for those hypernodes. As such, a decider compares x to $chosen[l_2'][l_1']$ for any value of l_1'. Listing 17 shows the modified decider code.

B.5 Summary

This completes the protocol derivation from the basic DCP to a solution for coordinate selection in ALIAS. L_1 switches function as choosers for L_1 coordinates (Listings 8 and 12), as potential representatives for L_2 coordinate selection (Listings 13, 15, and 16) and as hypernode calculators (Listing 10). L_2 switches act as relays for L_2 coordinate selection (Listing 14), as deciders for L_1 coordinate selection (Listing 6) and as hypernode change notifiers (Listing 9). Finally L_3 switches are deciders for L_2 coordinate selection (Listing 17).

Listing 17: Decider Algorithm (Distributed Chooser)

```
set⟨Switch⟩ $L_2$relays = ...
set⟨Switch⟩ $L_1$reps = ...
(set⟨Switch⟩)[$L_2$relays][$L_1$reps] choosers = ...
Choice[$L_2$relays][$L_1$reps] chosen = all[-1]
int[$L_2$relays][$L_1$reps] last_seq = all[0]

// when connected to new $L_2$ switch
F: when new $l_2 \in L_2$relays with representative $l_1$
    $L_2$relays ← $L_2$relays ∪ {$l_2$}
    $L_1$reps ← $L_1$reps ∪ {$l_1$}
    choosers[$l_2$][$l_1$] ← {$l_2$}
    chosen[$l_2$][$l_1$] ← -1
    last_seq[$l_2$][$l_1$] ← 0

// respond to a message from $L_2$ switch $l_2$
G: when receive ⟨s,x,hn,$l_1$⟩ from $l_2$
    $L_1$reps ← $L_1$reps ∪ {$l_1$}
    if $s \geq$ last_seq[$l_2$][$l_1$]
        foreach $l_2$' ∈ choosers[$l_2$][$l_1$] do
            choosers[$l_2$'][$l_1$] ← hn
            last_seq[$l_2$'][$l_1$] ← s
        if ∃ $l_2$'∈ $L_2$relays, $l_1$'∈ $L_1$reps: ($l_2$'∉ choosers[$l_2$][$l_1$]) ∧
                                        (chosen[$l_2$'][$l_1$'] == x)
            foreach $l_2$' ∈ choosers[$l_2$][$l_1$] do
                chosen[$l_2$'][$l_1$] ← -1
        else
            foreach $l_2$' ∈ choosers[$l_2$][$l_1$] do
                chosen[$l_2$'][$l_1$] ← x
    hints ← {chosen[$l_2$'][$l_1$']$_{\forall l_{1'} \in L_1reps,\ l_{2'} \in (L_2relays \setminus choosers[l_2][l_1])}$}
    hints ← hints \ {-1}
    send ⟨s,chosen[$l_2$][$l_1$],hints,$l_1$⟩ to $l_2$
```