# Bloom Filters

How I learned to stop worrying about errors and love memory efficient data structures

A presentation by Elliott Karpilovsky

# The Space and Time Impetuses

- "Set" data structures are used everywhere
  - Web caches, spellcheckers, databases, *etc*.
- The naïve implementation isn't efficient enough for systems applications, both space-wise and time-wise
- Using memory efficient data structures, can sacrifice a tiny bit of precision for incredible memory and run-time savings

# A Quick Review of Sets

- Mathworld:
  - **Set:** A set is a finite or infinite collection of objects in which order has no significance, and multiplicity is generally also ignored
  - **Multiset:** A set-like object in which order is ignored, but multiplicity is explicitly significant. Therefore, multisets {1, 2, 3} and {2, 3, 1} are equivalent, but {1, 1, 2, 3} and {1, 2, 3} differ

# A Quick Review of Sets

- A "C++" Set:
  - `add<T>(T item)`
  - `contains<T>(T item)`
  - `remove<T>(T item)`

- A "C++" Multiset additionally has:
  - `num_occurs<T>(T item)`

# A Special Note on “Remove”

- Will assume that `remove<T>` is only called on elements that are *actually* in the set
- Assumption is okay, since in many applications, all items in the set are stored “offline” and it is possible to check if an item truly is in the set

# A Naïve Set Implementation

- Assume:
  - Know, a priori, that the set will contain **n** elements
  - Each element consumes **m** bits of space
    - **n, m** may be extremely large

- Construct:
  - Use a balanced binary tree
    - total ordering always exists

# A Naïve Set: Time Analysis

- add: $O(m \log(n))$
- remove: $O(m \log(n))$
- contains: $O(m \log(n))$
- num_occurs: $O(m \log(n))$
  - "Set" can be converted to a "multiset" by extending each element with a counter

# A Naïve Set: Space Analysis

- O(mn) storage
  - Stores **n** elements, each **m** bits long
  - Assume counter size is negligible

# Can we do better?

- O(mn) storage required
- O(m log(n)) time for all operations
  - Not looking so hot for systems applications!

- A better approach: use buckets and hashes
  - Leads to the hash set data structure
  - Commonly used in systems applications

# Hash Set: Implementation

- Have a fixed array of size **q**
- Have a hash function that maps elements between 0 and q-1
- Use linked lists to store elements that hash to the same value
- See any standard reference (*i.e., C.L.R.S.*) for implementation details

# Hash Set: Time Analysis

- Define the **load factor** $\alpha = n/q$
- For **n** elements, expected number of items in each bucket is $\alpha$
- Takes O(m) time to hash
- Takes $O(m\alpha)$ time, on average, to search a bucket

# A Hash Set: Time Analysis

- add: $O(m(1 + \alpha))$
- remove: $O(m(1 + \alpha))$
- contains: $O(m(1 + \alpha))$
- num_occurs: $O(m(1 + \alpha))$
  - "Set" can be converted to a "multiset" by extending each element with a counter

# A Hash Set: Space Analysis

- O(mn) storage
  - Stores **n** elements, each **m** bits long
  - Assume counter size is negligible
- Additional O(n/$\alpha$) storage for linked lists
  - Generally negligible relative to O(mn)

# A Comparison: Hash vs. Naïve

| | Memory | Runtime |
|---|---|---|
| **Hash set** | O(mn) | $O(m(1 + \alpha))$ |
| **Naïve set** | O(mn) | O(m log(n)) |

# Are we stuck with O(mn) Space?

- Could it be that there's no way around it?
  - Indeed, we are stuck…

# Are we stuck with O(mn) Space?

- Could it be that there's no way around it?
  - Indeed, we are stuck… but only if we want an error rate of zero

# Are we stuck with O(mn) Space?

- Could it be that there’s no way around it?
  - Indeed, we are stuck… but only if we want an error rate of zero

- What if we’re willing to tolerate a small error rate?
  - In this case, there is a solution!

# Bloom Filters to the Rescue

- Unlike hash sets, Bloom Filters:
  - Have a fixed error rate
  - Use memory linear in **n**
  - Have runtime linear in **m**
- Very easy to implement
- Will never report false-negatives

# The Motley Bloom Filter Crew

- Standard Bloom Filter
  - Supports `add<T>, contains<T>`
- Counting Bloom Filter
  - Supports `remove<T>`
- Spectral Bloom Filter
  - Supports `num_occurs<T>`
- Other Variants
  - Compressible Bloom Filter, External Memory Filters, Bloomier Filters, etc.

# Bloom Filter: Implementation

- Start off with a bit array of size **q**, initializing all bits to 0
- Create **k** different hash functions $h_1, h_2, \ldots, h_k$
  - Can create using SHA-1 and randomly salt
  - Hash to values between **0** and **q-1**
  - Assume negligible storage requirements for the hash functions

# Bloom Filter: Implementation

- When we want to add an element, hash it **k** times and set the corresponding bits to 1

```
add<T>(T item)
{
  for(int i = 0; i < k; i++)
      array[h_i(item)] = 1;
 }
```

# Bloom Filter: Implementation

- When we want to check for containment, hash **k** times and see if all **k** bits are set to 1

```
contains<T>(T item)
{
for(int i = 0; i < k; i++)
  if(!array[h_i(item)]) return false;

return true;
}
```

# Bloom Filters: Analysis

- Memory usage is O(q)
  - *q* is any value we pick
- Runtime for all operations is O(mk)
  - *k* is any value we pick
- Error rate is completely determined by our choices of *q* and *k*

# Bloom Filters: Error Analysis

- How should we choose **q**?
- How should we choose **k**?
- What should we do to minimize the error?

# Bloom Filters: Error Analysis

- The probability of a bit still being 0 after all **n** elements are inserted is:
  - $p = (1 - 1/q)^{kn} \approx e^{-kn/q}$
- The probability of a false positive is then:
  - $f = (1 - p)^k$

# Bloom Filters: Error Analysis

- Want to minimize: $f = e^{(k \ln(1 - p))}$
  - Assume that **q** and **n** are fixed, solve for **k**
- Minimizing **k ln(1 – p)** also minimizes **f**
- Same as minimizing: $-q/n \ln(p) \ln(1-p)$
  - $k = -q/n * \ln(e^{-kn/q})$
  - $p = e^{-kn/q}$

# Bloom Filters: Error Analysis

- Minimize: $-q/n * \ln(p) \ln(1-p)$
- By symmetry, has global minimum at $p = ½$
- Corresponds to $k = \ln 2 * (q/n)$
  - $k = -q/n \ln(p)$
  - $k = \ln(1/p) * (q/n)$

# Bloom Filters: Error Analysis

- When $k = \ln 2 * (q / n)$, false positive rate becomes:
  - $f = (1/2)^k \approx (0.6185)^{q/n}$
- By letting $q = cn$, the rate becomes:
  - $f \approx (0.6185)^c$
  - $f \approx 2.14\%$ for $c = 8$
  - $f \approx 0.05\%$ for $c = 16$

# Bloom Filters: In Practice

- Memory usage is **O(cn)**
  - Compare to O(mn) for naïve sets, hash sets
- Runtime is **O(cm)**, since k = ln(2) * c
  - Compare to O(m log(n)) for naïve sets
  - Compare to O(m (1 + $\alpha$)) for hash sets
- Error rate is $(.6815)^c$
  - Compare to 0 for naïve sets, hash sets

# Can we do better than Bloom?

- Is it possible to get better memory savings than Bloom Filters?
  - Yes and no

- For a given error rate, Bloom Filters are within a factor of 1.44 of the theoretically most optimal data structure
  - However, Bloom Filter implementations are exactly the same for any set of objects
  - Not known how to implement the theoretically most optimal structure

# An Aside: Bloom Filter Regalia

- Ever…
  - Wanted to make small chat by the watercooler?
  - Needed to entertain a kid's birthday party?
- But couldn't find an interesting topic?
- Amaze and dazzle your friends and colleagues with Bloom Filter Tricks!

# Party Tricks: Bloom Union

- Want to take the union of two bloom filters that have the same hash functions?
- Just OR all the bits together!

# Party Tricks: Bloom Shrink

- Want to cut memory usage in half?
- OR the first half of the array with the second half!
- Mask the high order bit on your hash functions
  - Side effect: error rate will increase

# Counting Bloom Filters

- Very slight modification of the Bloom Filter
  - Adds support for remove<T>

- Instead of using a bit array, use a counter array

# Counting Bloom Filters

```
add<T>(T item)
{
  for(int i = 0; i < k; i++)
      array[h_i(item)]++;
 }
```

# Counting Bloom Filters

```
contains<T>(T item)
{
  for(int i = 0; i < k; i++)
    if(!array[hi(item)])
         return false;

  return true;
}
```

# Counting Bloom Filters

```
remove<T>(T item)
{
  for(int i = 0; i < k; i++)
        array[h_i(item)]--;
}
```

# Counting Bloom Filters

- Memory usage is now O(qt), where *t* is the size of the counter in bits
- How large should we set *t* ?
  - Assume that the data is "uniform"
  - Doing some calculations, the probability that any counter will exceed value *j* is:

    $\text{Prob(any counter} \geq j) < q\ (1.885\ /\ j)^j$

# Counting Bloom Filters

- Prob(overflow) < q $(1.885 / j)^j$
  - $t = 2 \rightarrow j = 4$, Prob < 0.049 q
    - If c = 8, then even for two items, bound is bad (~.78)
  - $t = 3 \rightarrow j = 8$, Prob < 0.0000095 q
    - If c = 8, then bound becomes bad if we store more than a thousand items (~.08)

# Counting Bloom Filters

- Prob(overflow) < q $(1.885 / j)^j$
  - $t = 4 \rightarrow j = 16$, Prob < $1.38 * 10^{-15} * q$
    - If c = 8, good bound, even if you expect over a billion items (~.000011)

# Counting Bloom Filters

- Prob(overflow) < $q\ (1.885 / j)^j$
  - $t = 5 \rightarrow j = 32$, Prob < $4.4 * 10^{-40} * q$
  - $t = 6 \rightarrow j = 64$, Prob < $1.06 * 10^{-98} * q$
  - $t = 7 \rightarrow j = 128$, Prob < $3.29 * 10^{-235} * q$

# Counting Bloom Filters

- Prob(overflow) < q $(1.885 / j)^j$
  - $t = 8 \rightarrow j = 256$, Prob < $9.34 * 10^{-547} * q$
  - Even if:
    - c = number of atoms in universe
    - n = number of atoms in the universe
    - q = cn = square of number of atoms in the universe
  - Probability of an overflow is about $10^{-350}$

# Spectral Bloom Filters

- Essentially, exactly the same as a Counting Bloom Filter
  - Adds support for `num_occurs<T>`
  - Runs in O(mk) time, like all other operations
  - Error rate is exactly the same as the standard Bloom Filter error rate: $(1 - p)^k$

# Spectral Bloom Filters

The minimum selection estimator:

```
num_occurs<T>(T item)
{
  int smallest = overflow_value;
  for(int i = 0; i < k; i++)
     if(array[h_i(item)] < smallest)
         smallest = array[h_i(item)];

  return smallest;
}
```

# Compressible Bloom Filters

- Could imagine:
  - “Zipping” the bit array when not in use
  - “Unzipping” the bit array when an operation is called
  - “Re-zipping” it afterward

- Would slow down the program, but could save even more memory

- Is this possible?

# Compressible Bloom Filters

- With the standard Bloom Filter, no!
  - Remember, p = ½ when $k = c \ln(2)$ , so each bit has a ½ chance of being a 1
  - Essentially a random stream of 1's and 0's

# Compressible Bloom Filters

- What if we…
  - Reduce the number of hash functions (less hashing means more zeroes)
  - Increase the size of the array (to compensate for the increased error rate)
  - Then try compression
- Will the new filter be smaller and have about the same error rate?

# Compressible Bloom Filters

- Surprisingly, yes!
  - Example taken from Broder's survey paper
  - q is size of the array (uncompressed)
  - z is the size of the array (after compression)
  - f is the false-positive error rate

| | | |
|---|---|---|
| q/n | 16 | 48 |
| k | 11 | 3 |
| z/n | 16 | 15.829 |
| f | 0.000459 | 0.000222 |

# Other Bloom Filter Variants

- External Memory Filters
  - If the filter is too large to fit in memory, have a separate hash function decide what *section* of the array to search, and then perform the multiple hashing
    - Very slight increase in error rate

# Other Bloom Filter Variants

- Bloomier Filters
  - Create a lossy map from a domain **D** to a set **S**
    - "Near optimal" solution involves using multiple Bloom filters to represent each value in **S**
    - "Optimal" solution involves one-time construction of large lookup tables

# Applications: IP Traceback

- Suppose we find a malicious packet in our server log, want to find out where it came from
- Can’t trust the packet’s metadata

# Applications: IP Traceback

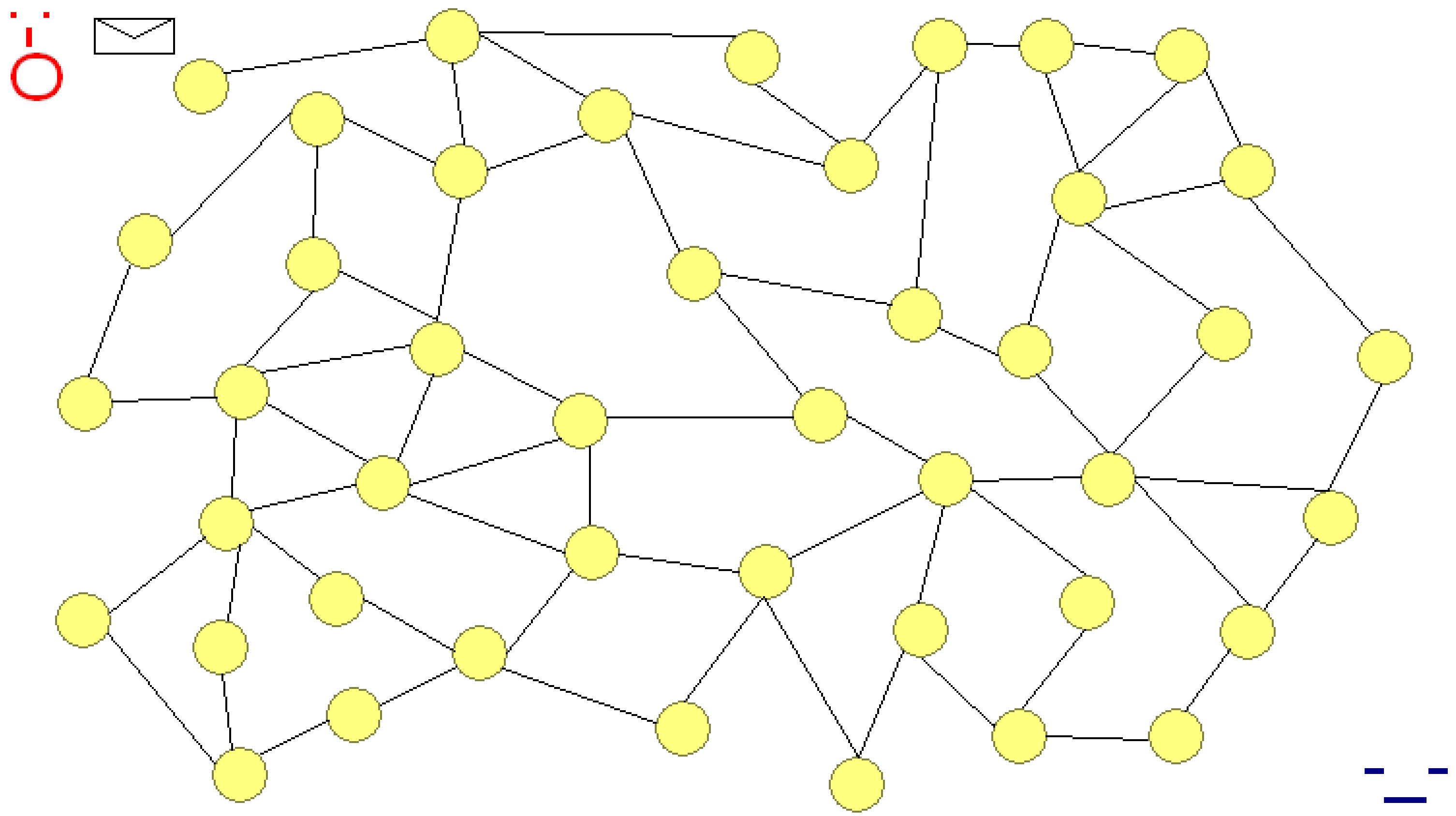
- Idea: Have every router keep a log of every packet it’s ever seen!

- Not that great, since routers see so many packets, have such limited memory, and must operate at breakneck speeds, they cannot possibly store this information

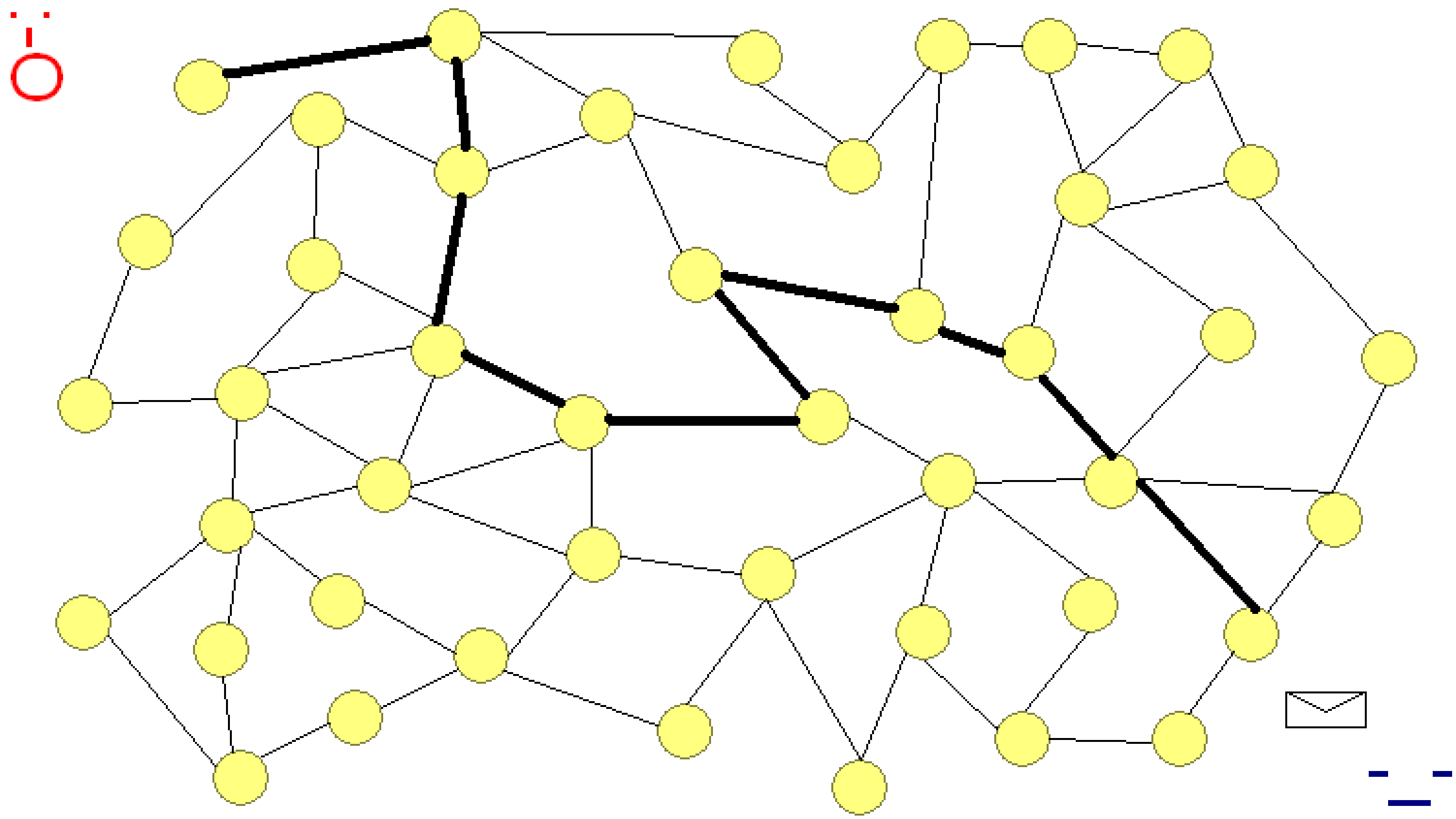
# Applications: IP Traceback

- Better Idea: Have every router keep a Bloom Filter of which packets its seen
- Query every router that the packet may have come from, see which ones saw the packet, have them recursively query routers they talk to, *etc*.
- Much more feasible, since Bloom Filters are fast and memory efficient

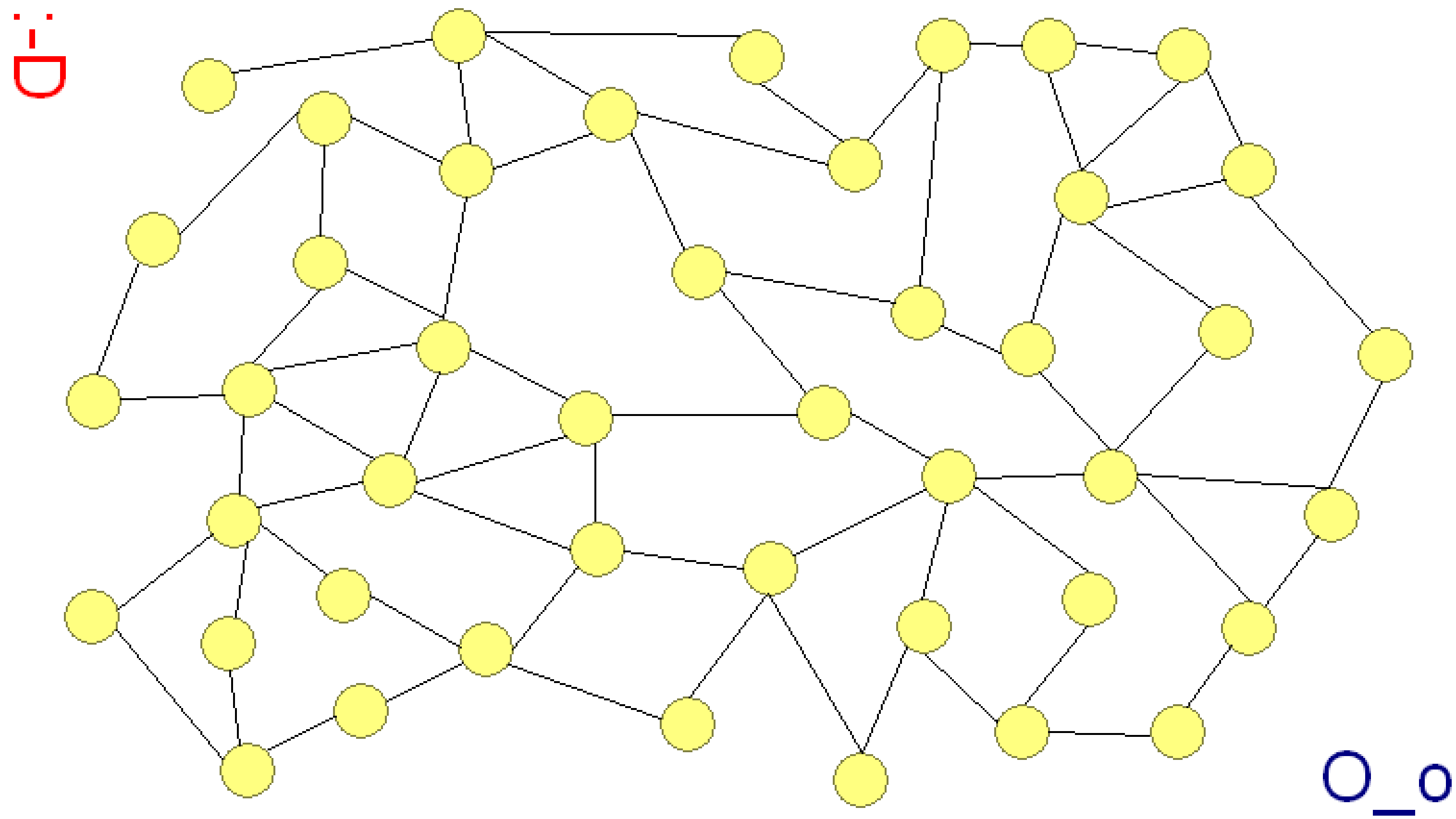
# Applications: IP Traceback



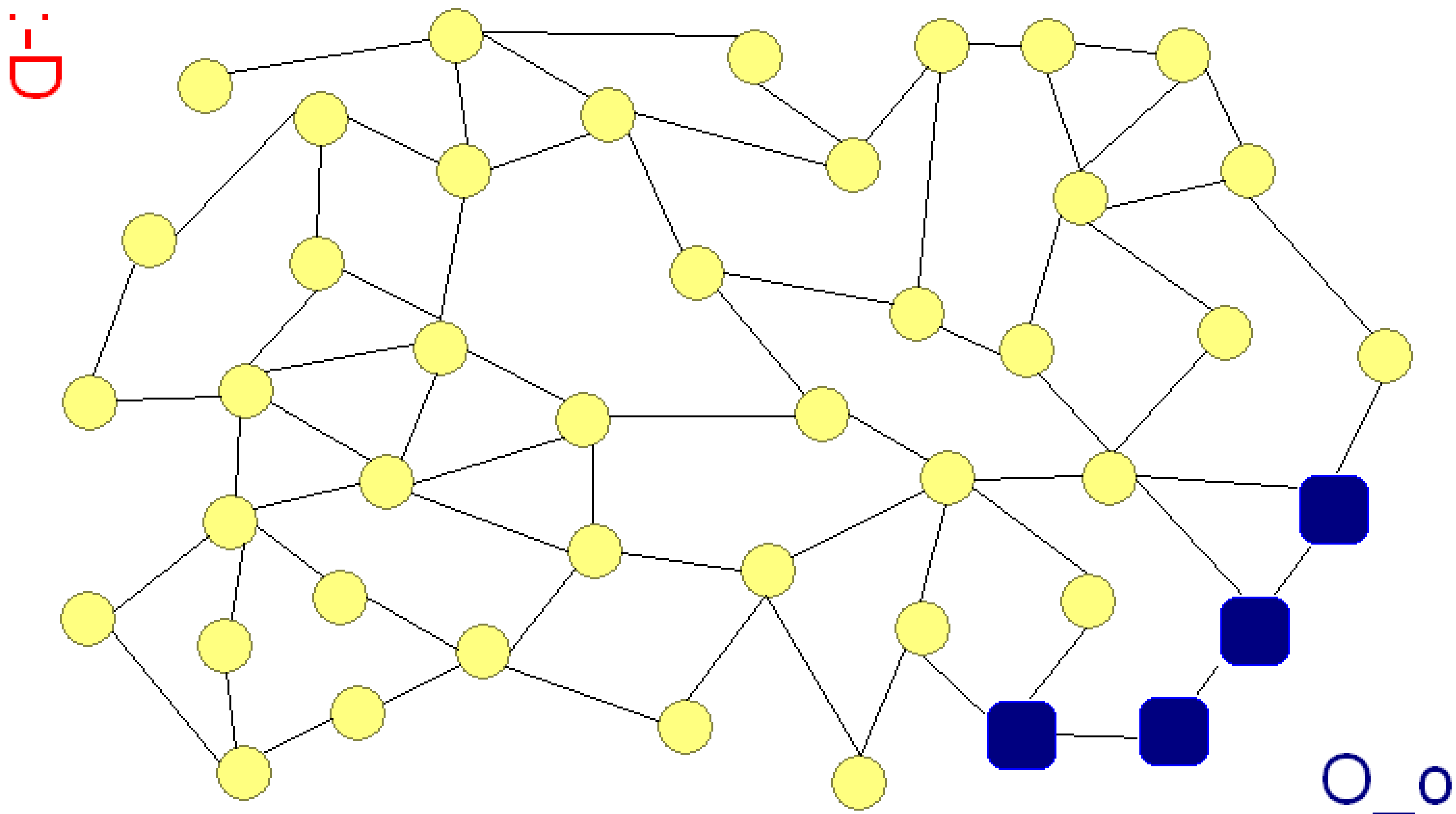
# Applications: IP Traceback



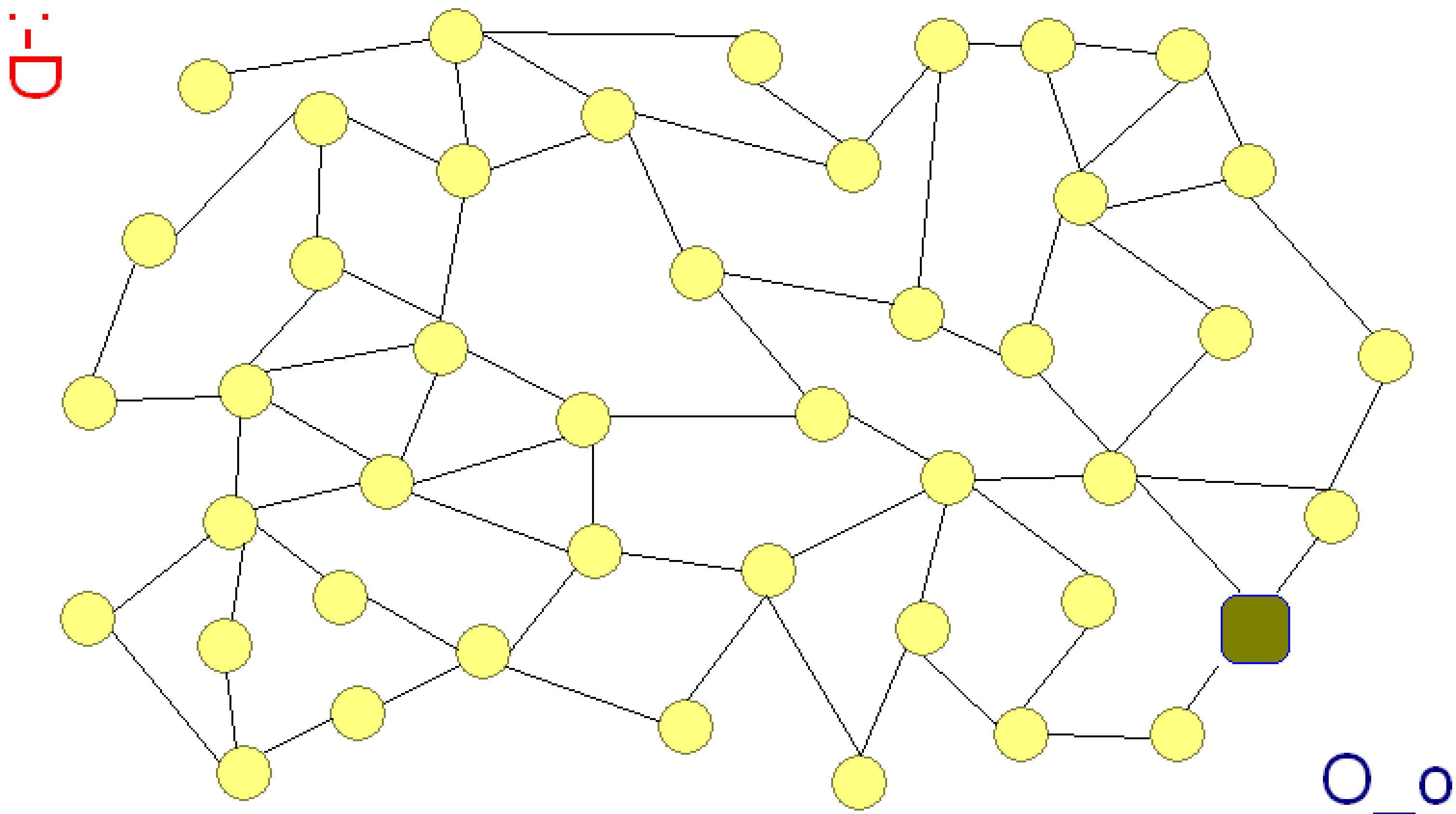
# Applications: IP Traceback



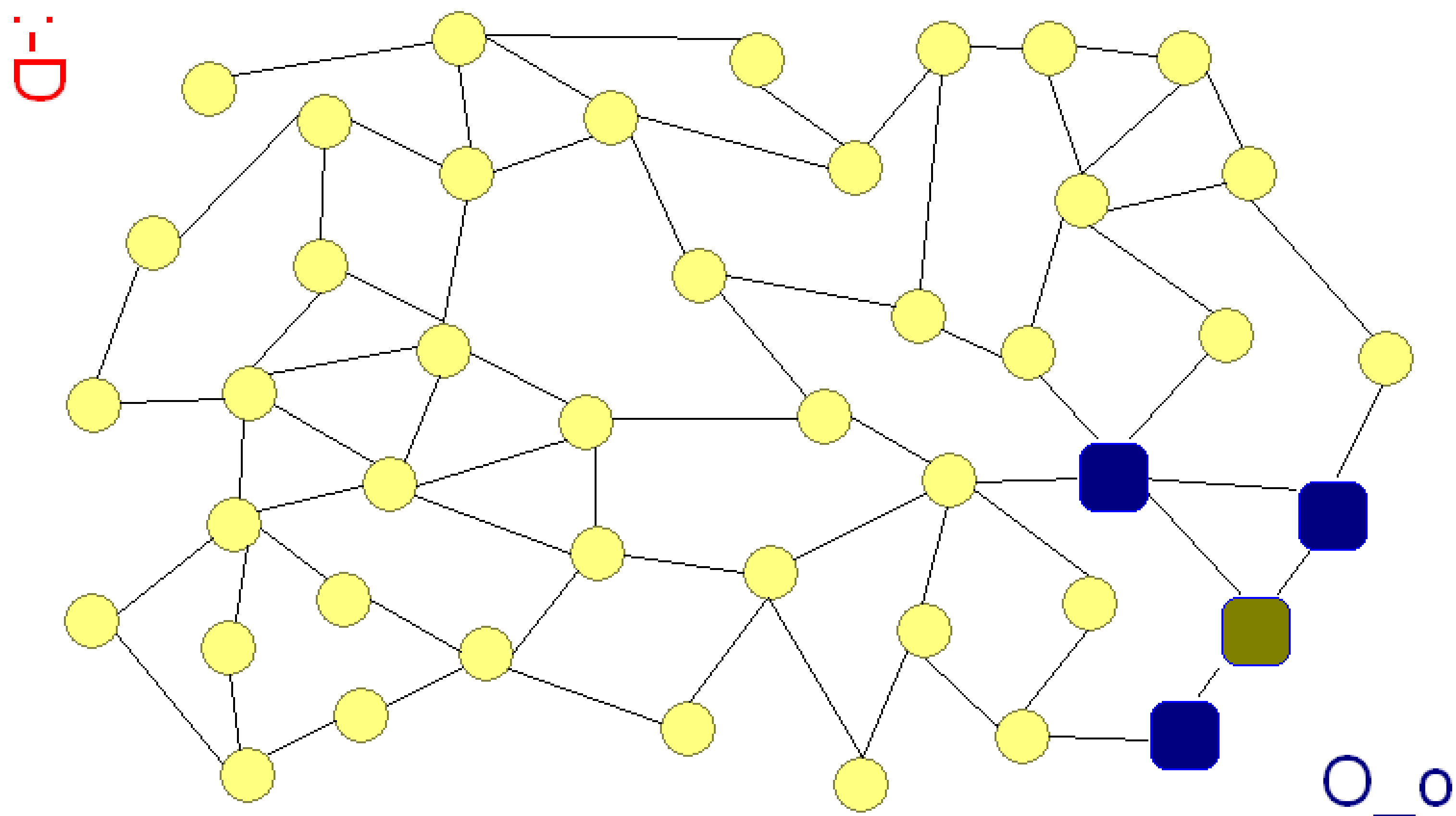
# Applications: IP Traceback



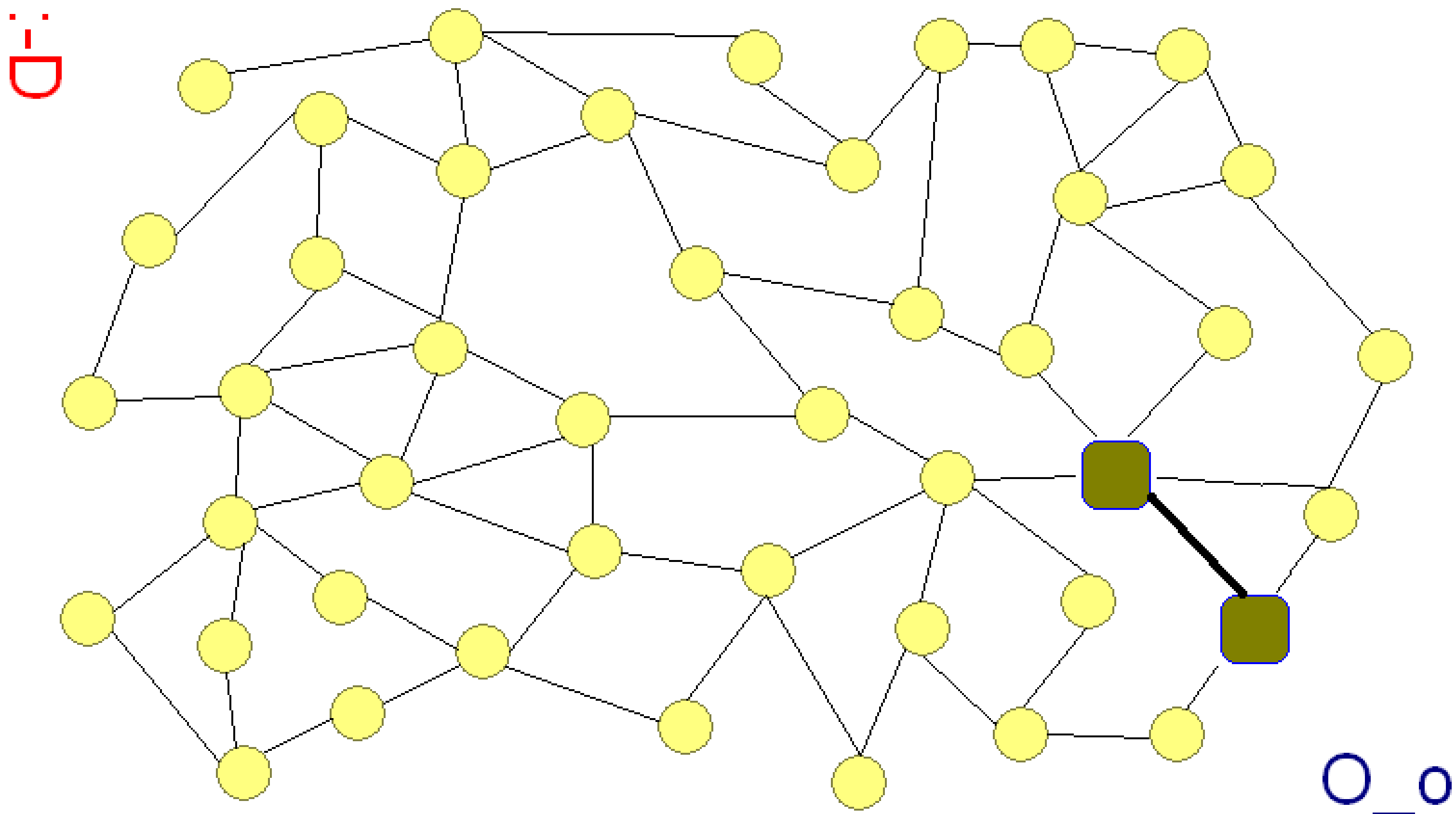
# Applications: IP Traceback



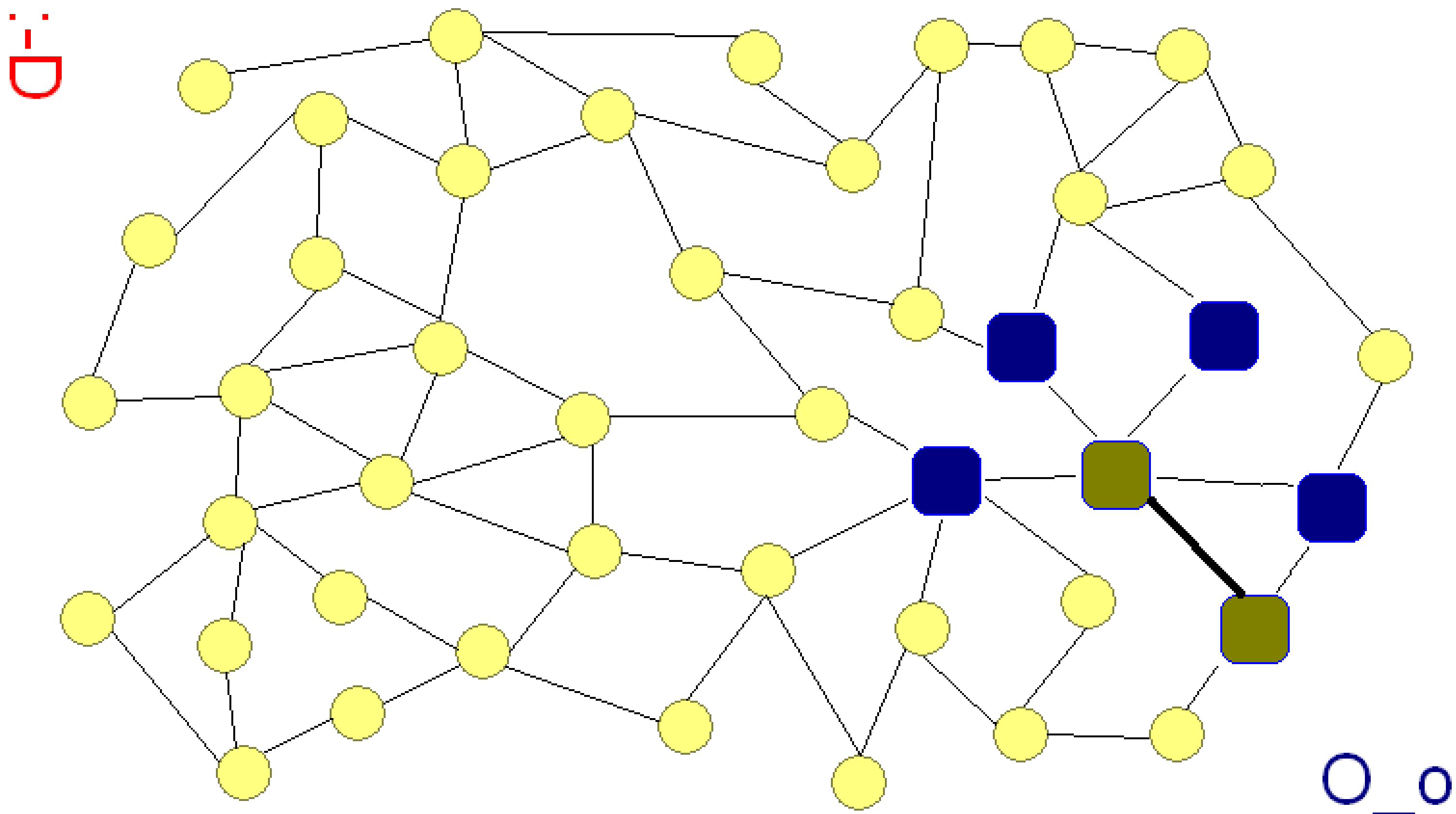
# Applications: IP Traceback



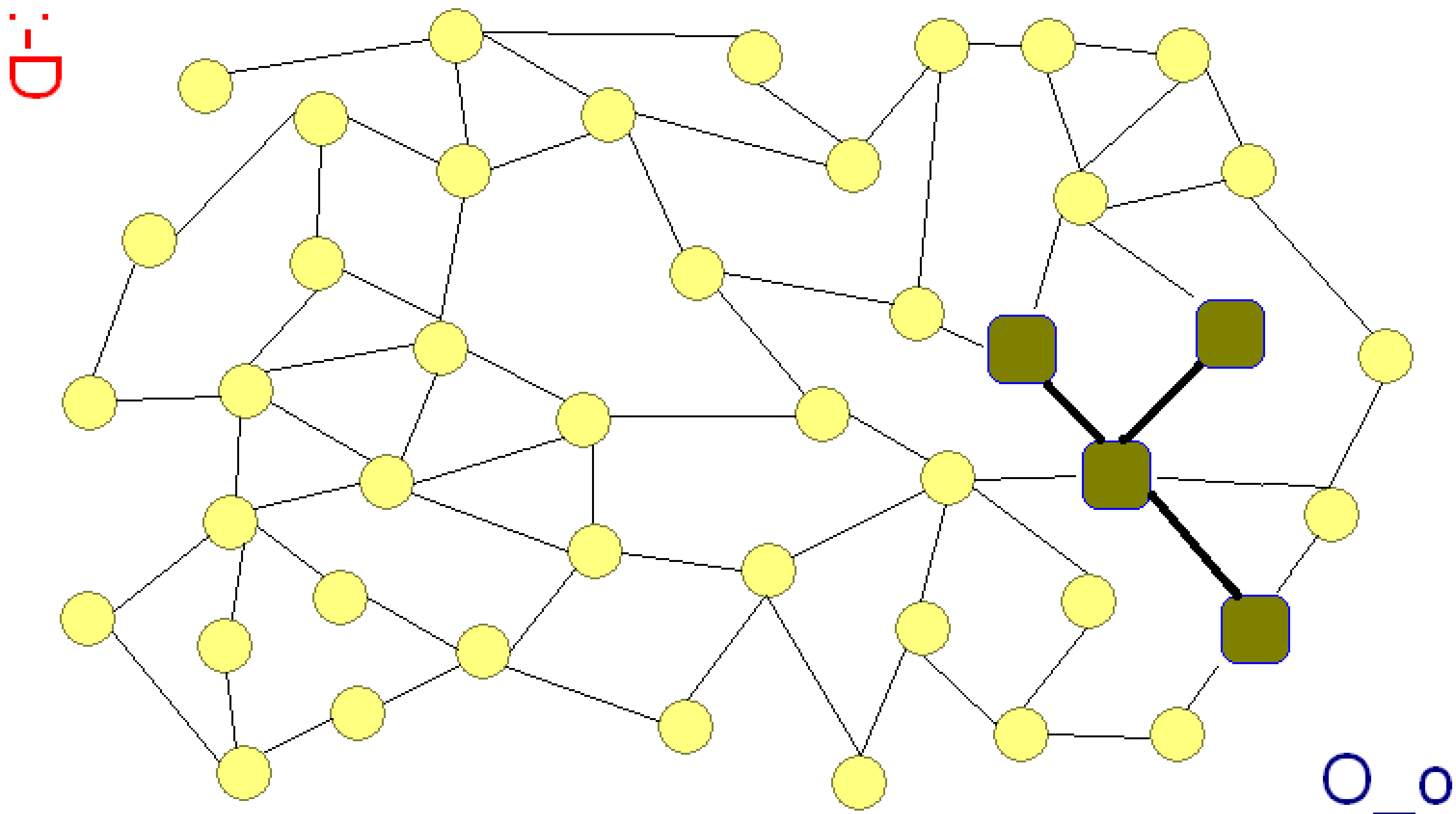
# Applications: IP Traceback



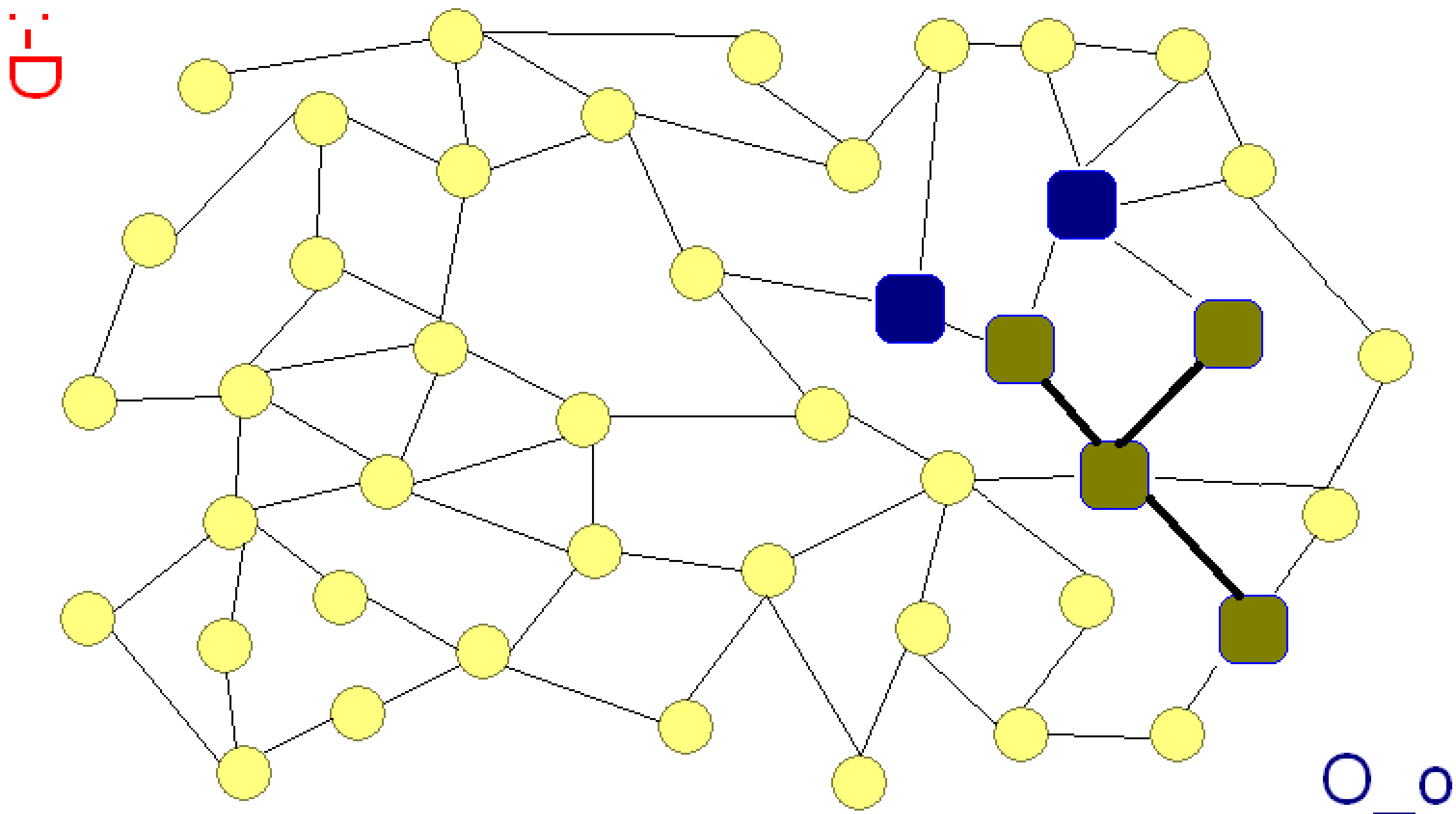
# Applications: IP Traceback



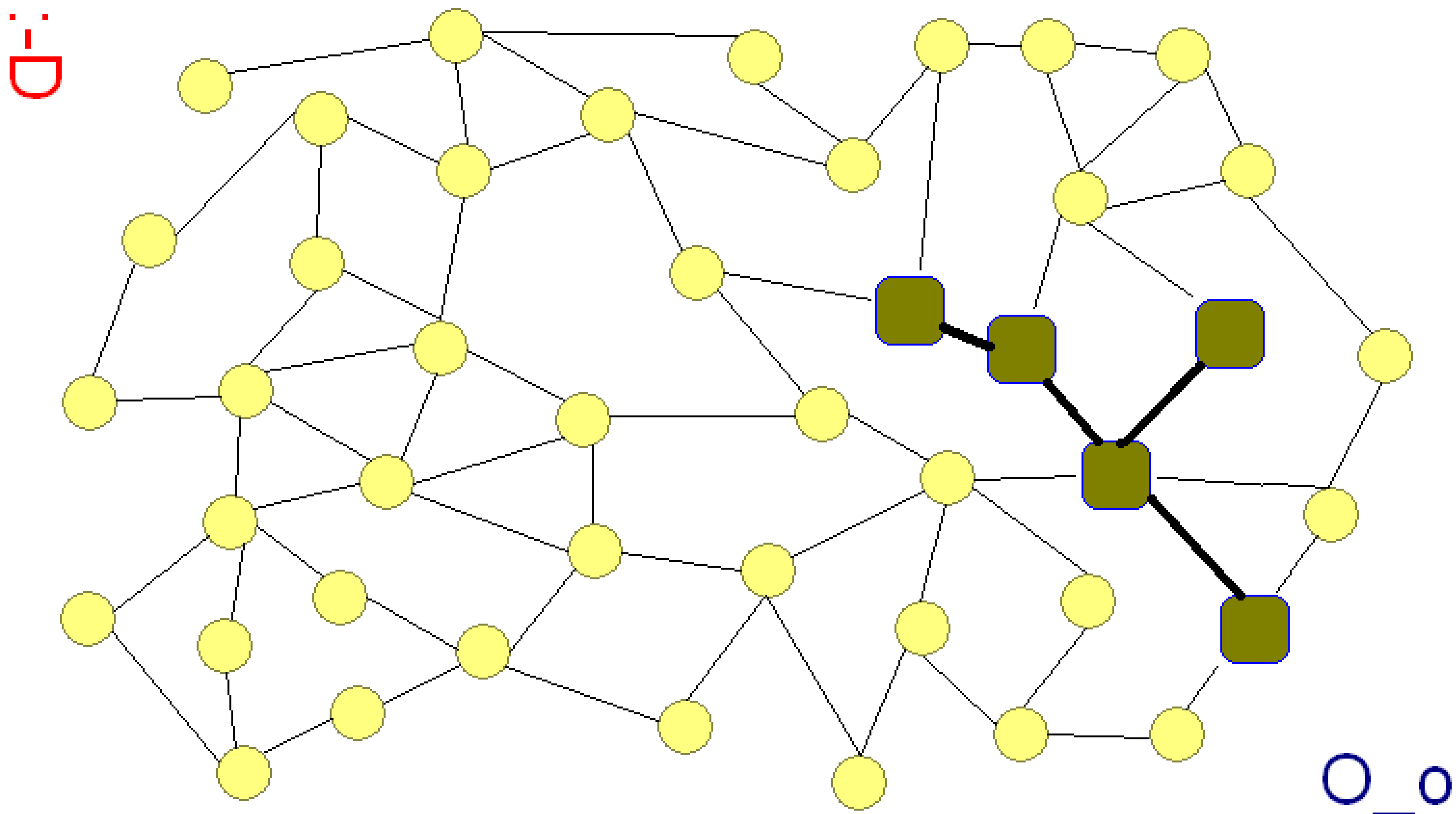
# Applications: IP Traceback



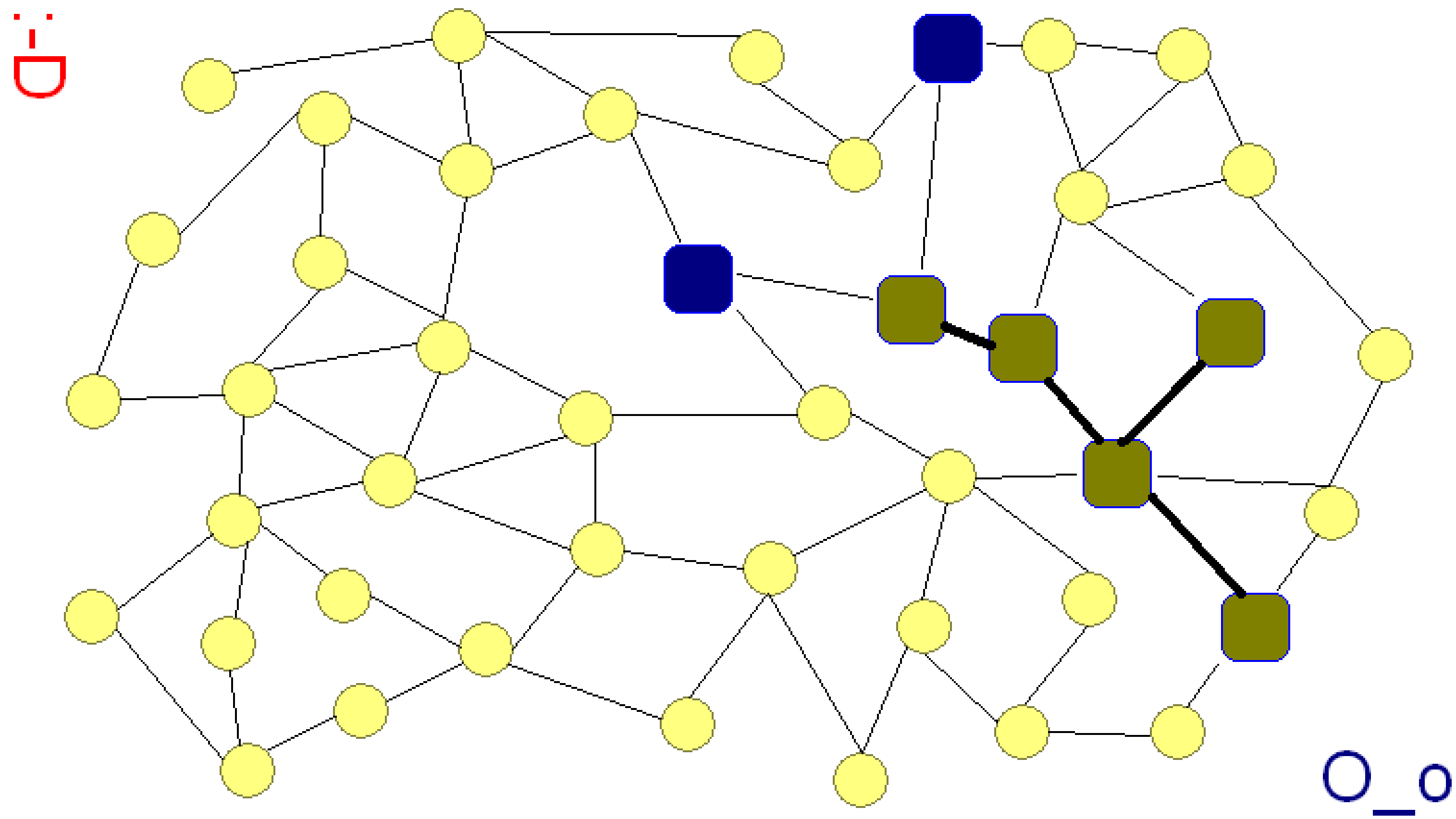
# Applications: IP Traceback



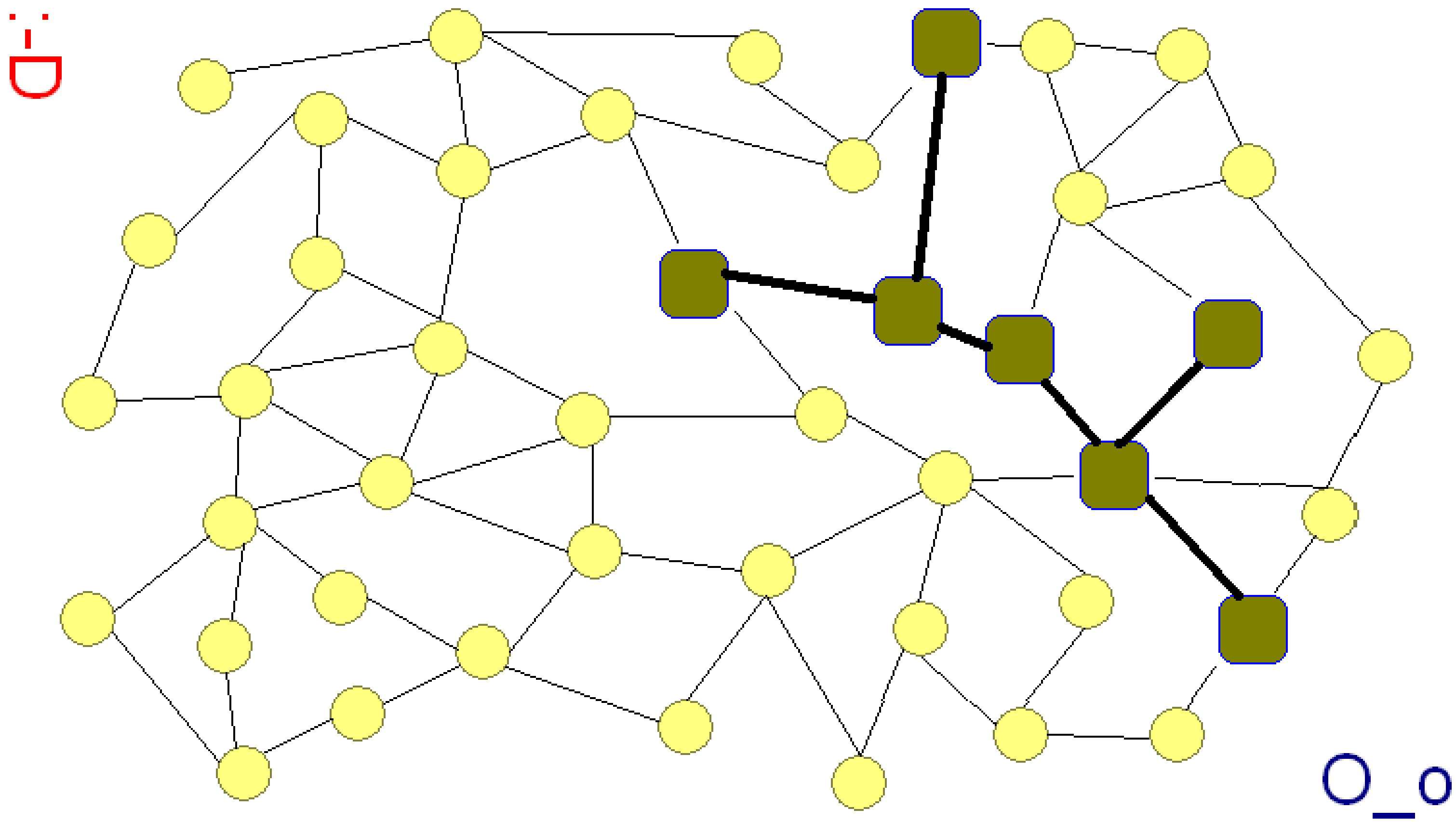
# Applications: IP Traceback



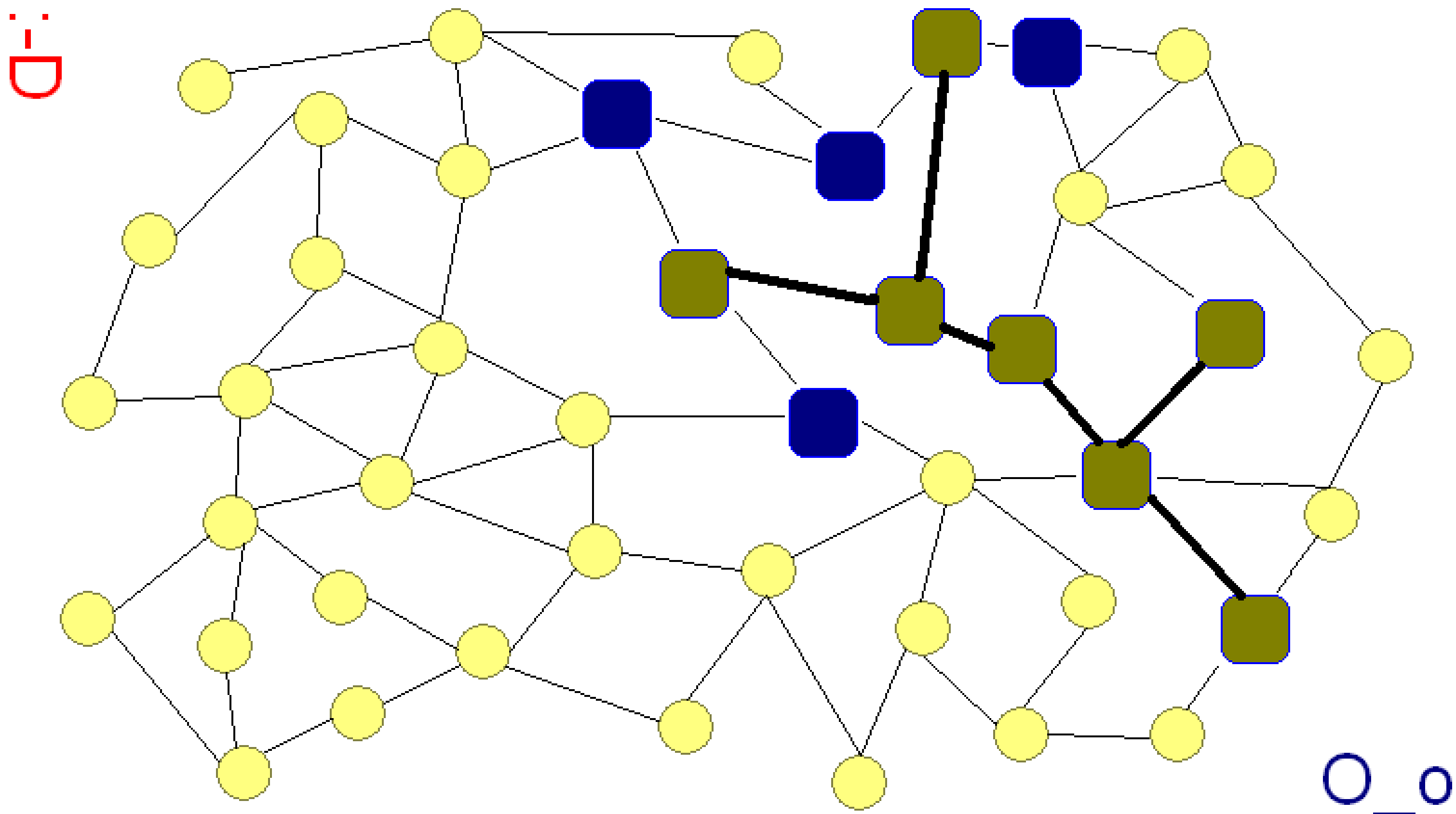
# Applications: IP Traceback



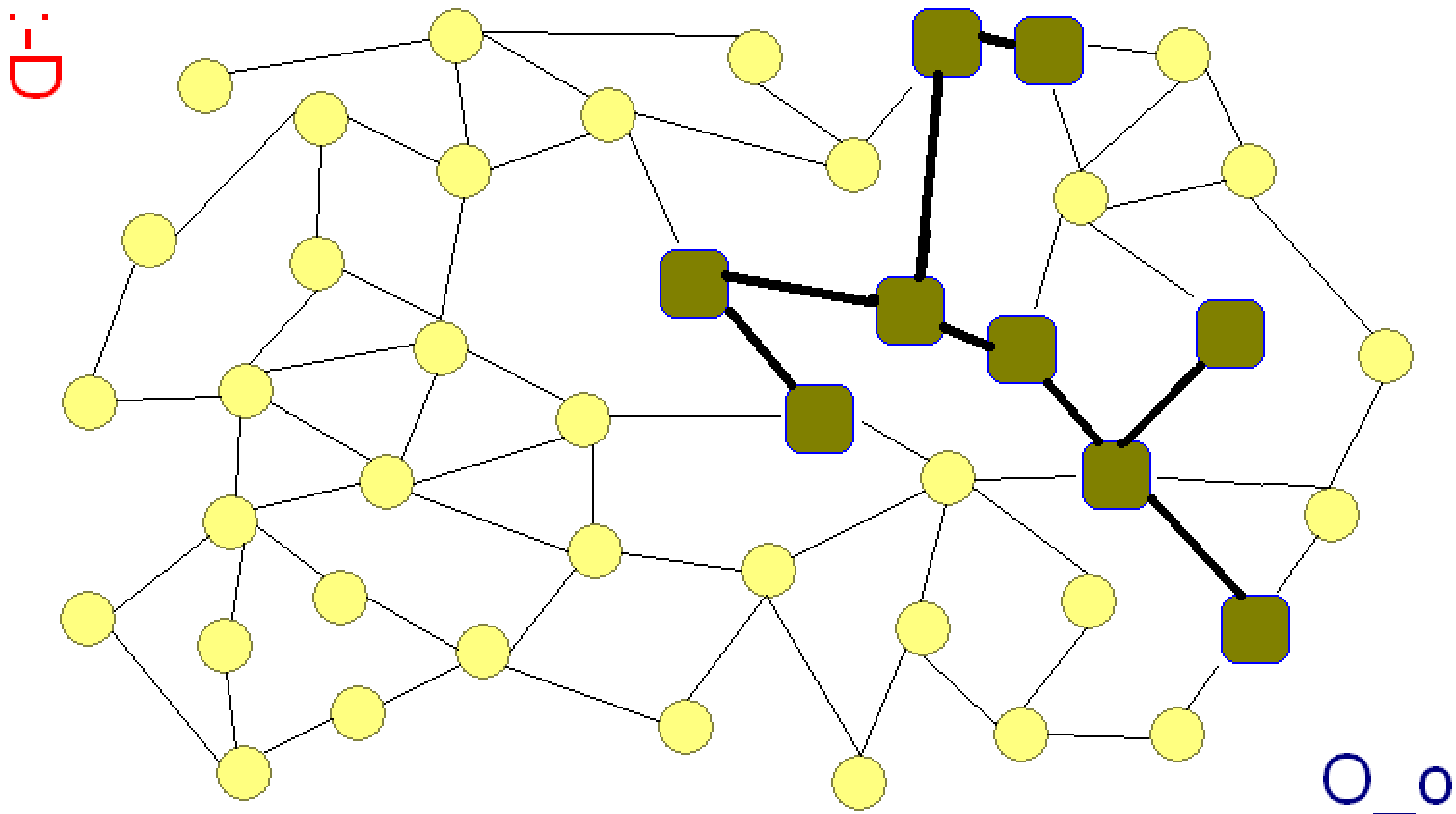
# Applications: IP Traceback



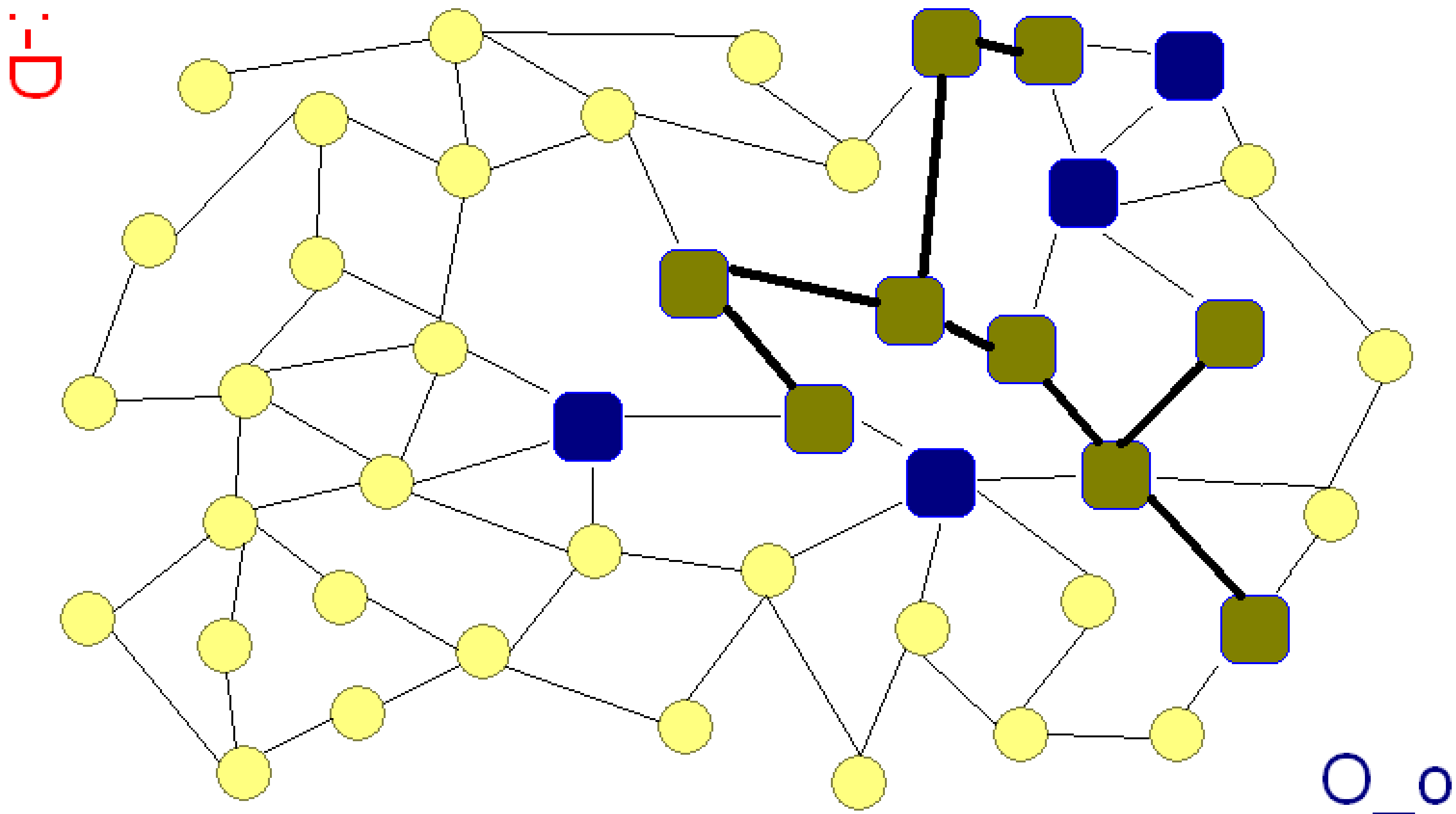
# Applications: IP Traceback



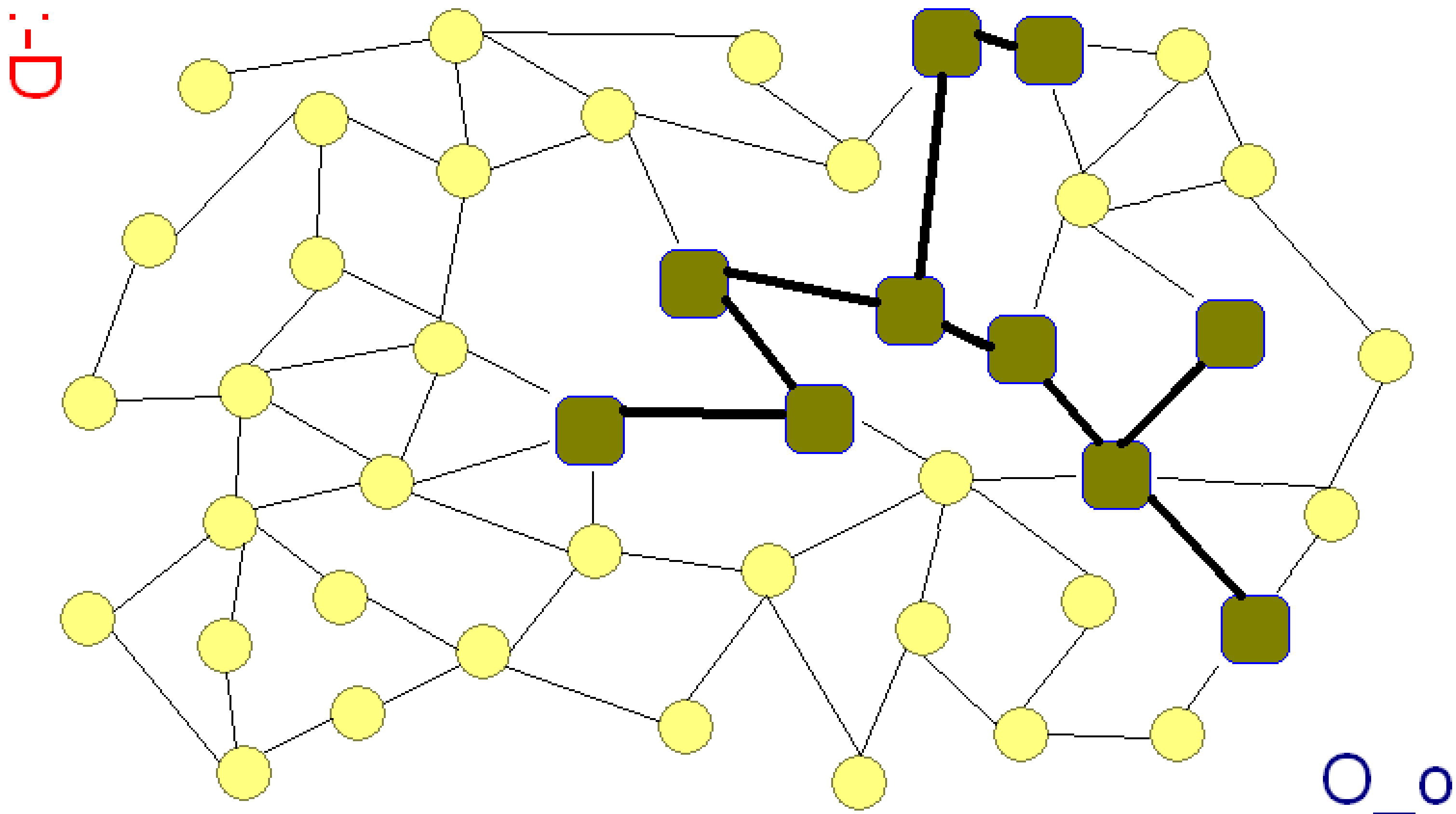
# Applications: IP Traceback



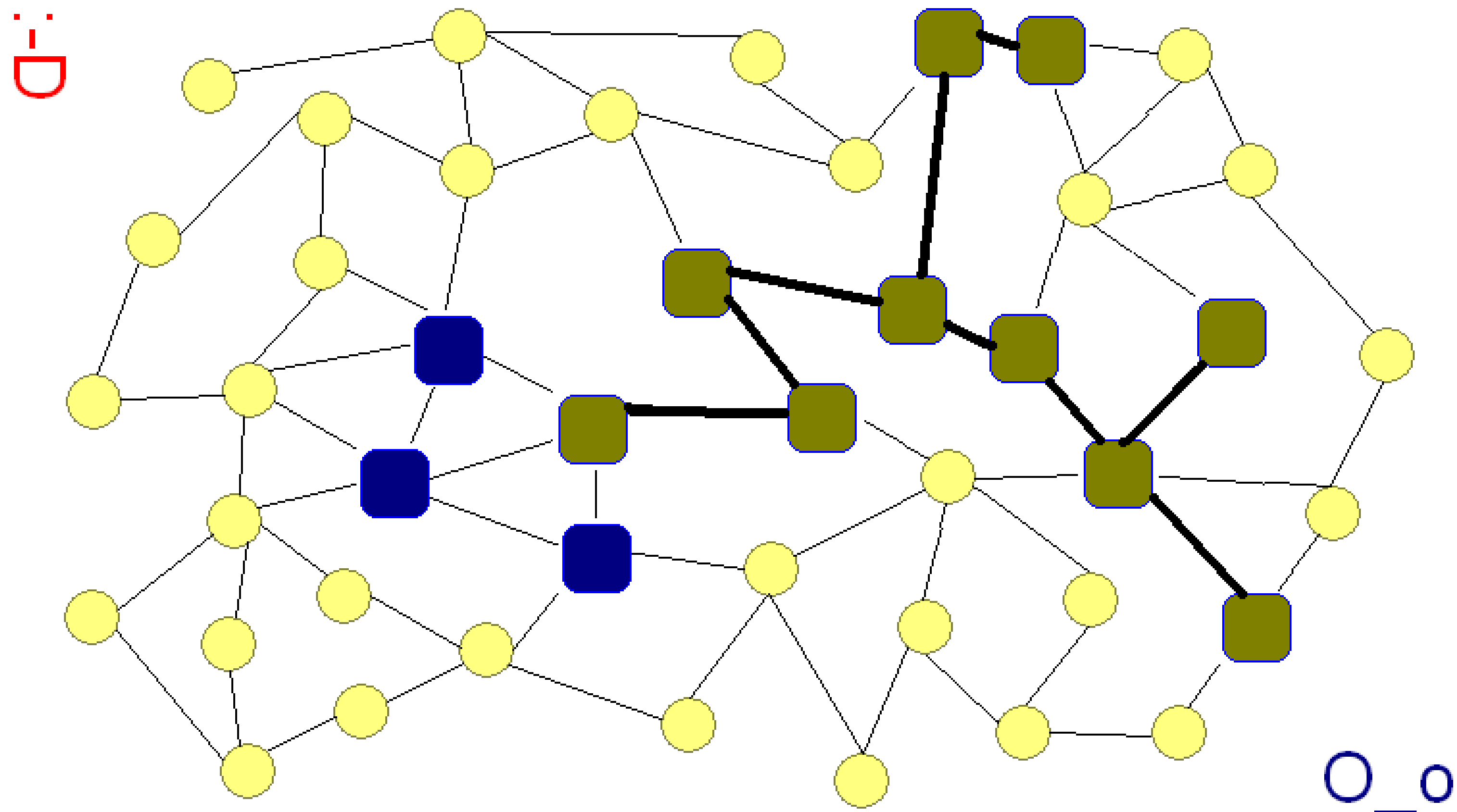
# Applications: IP Traceback



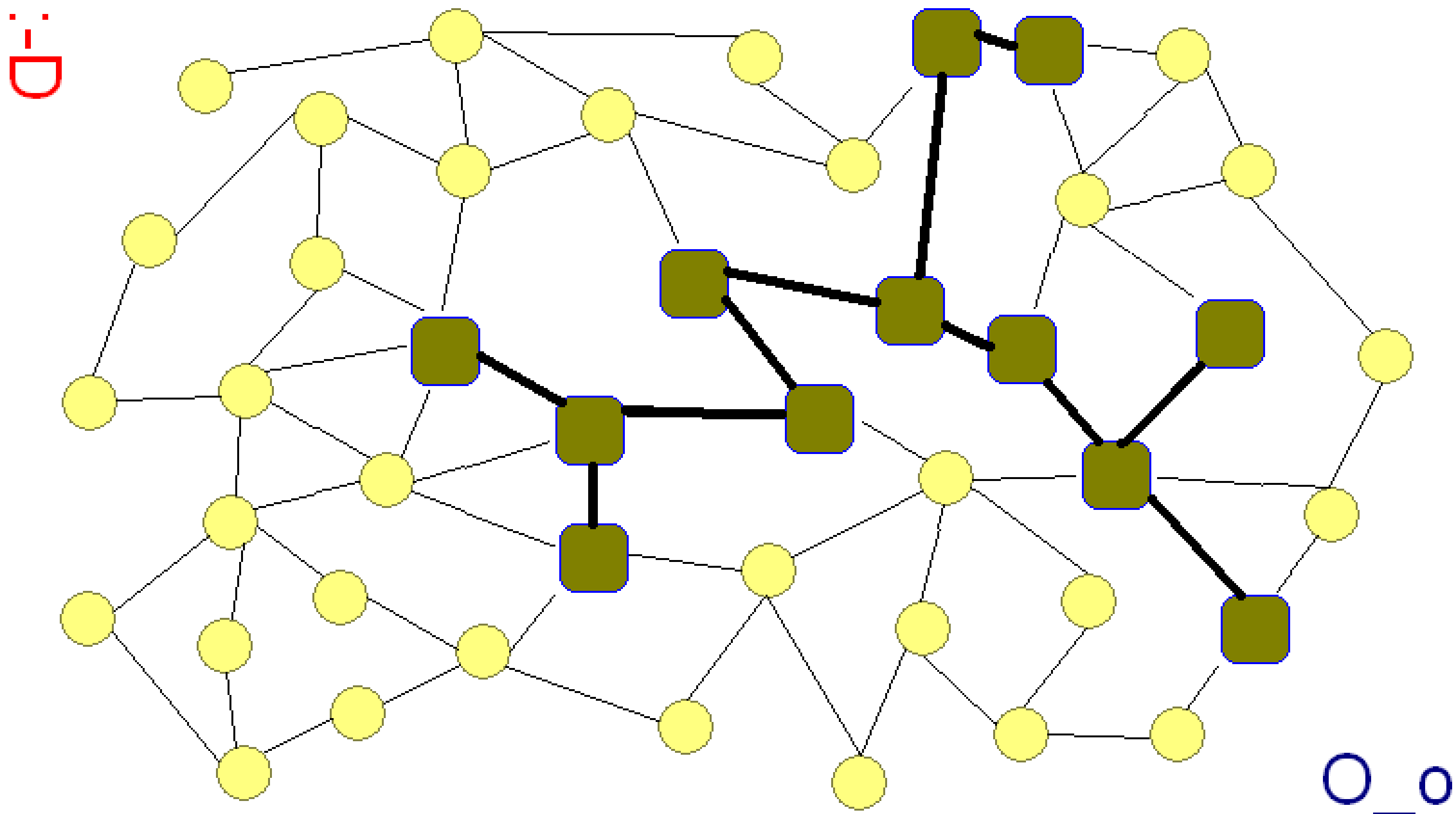
# Applications: IP Traceback



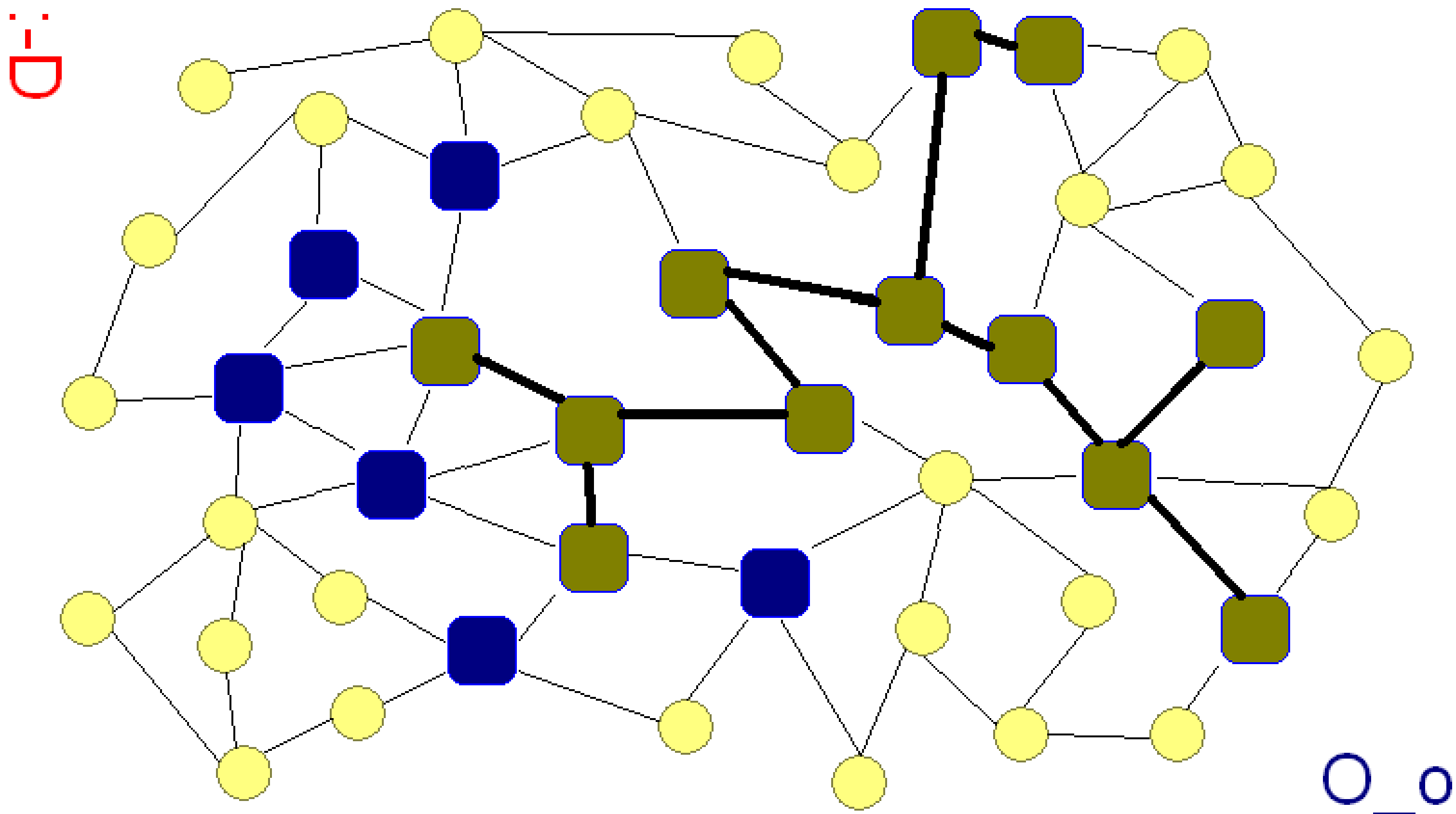
# Applications: IP Traceback



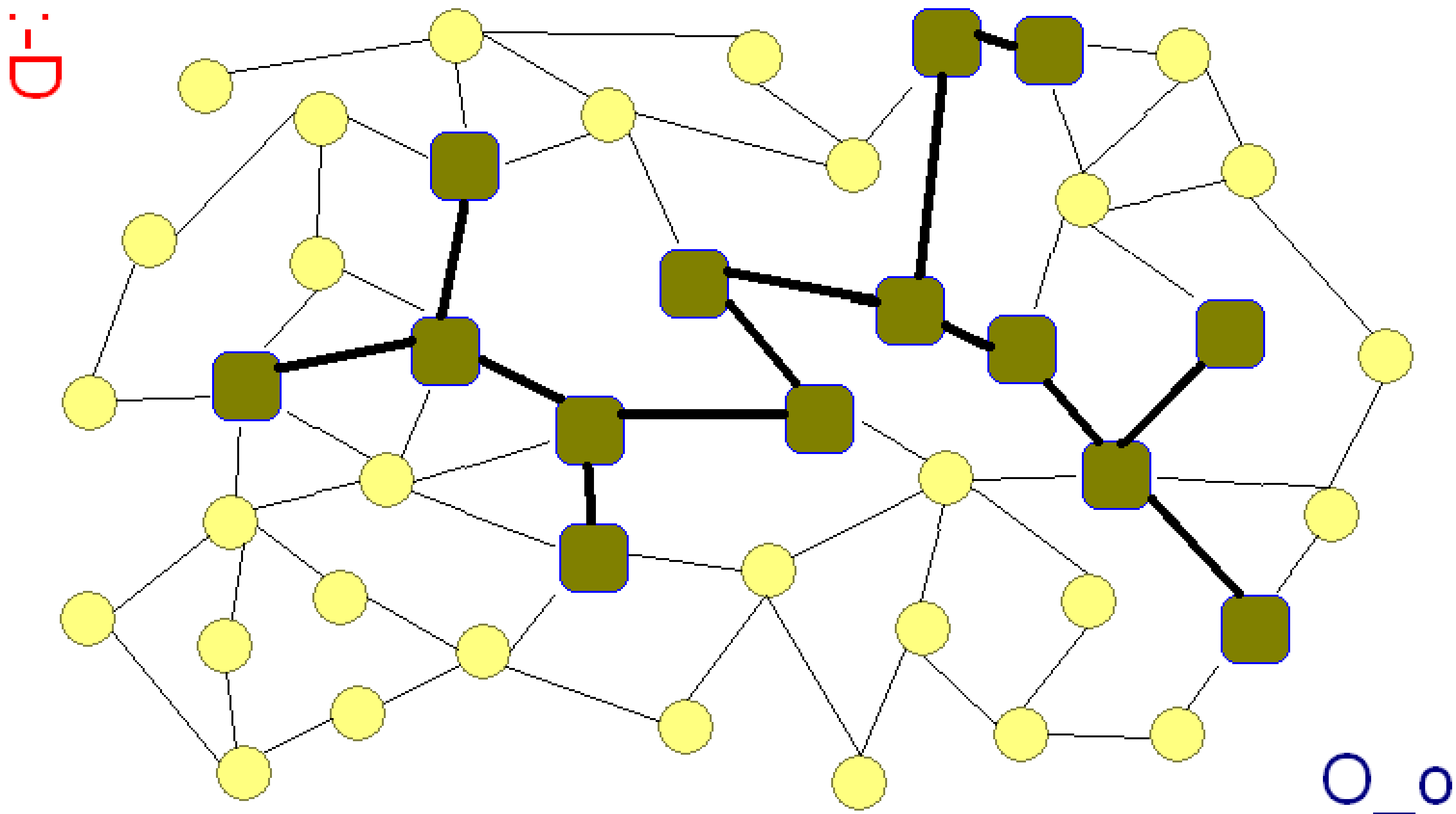
# Applications: IP Traceback



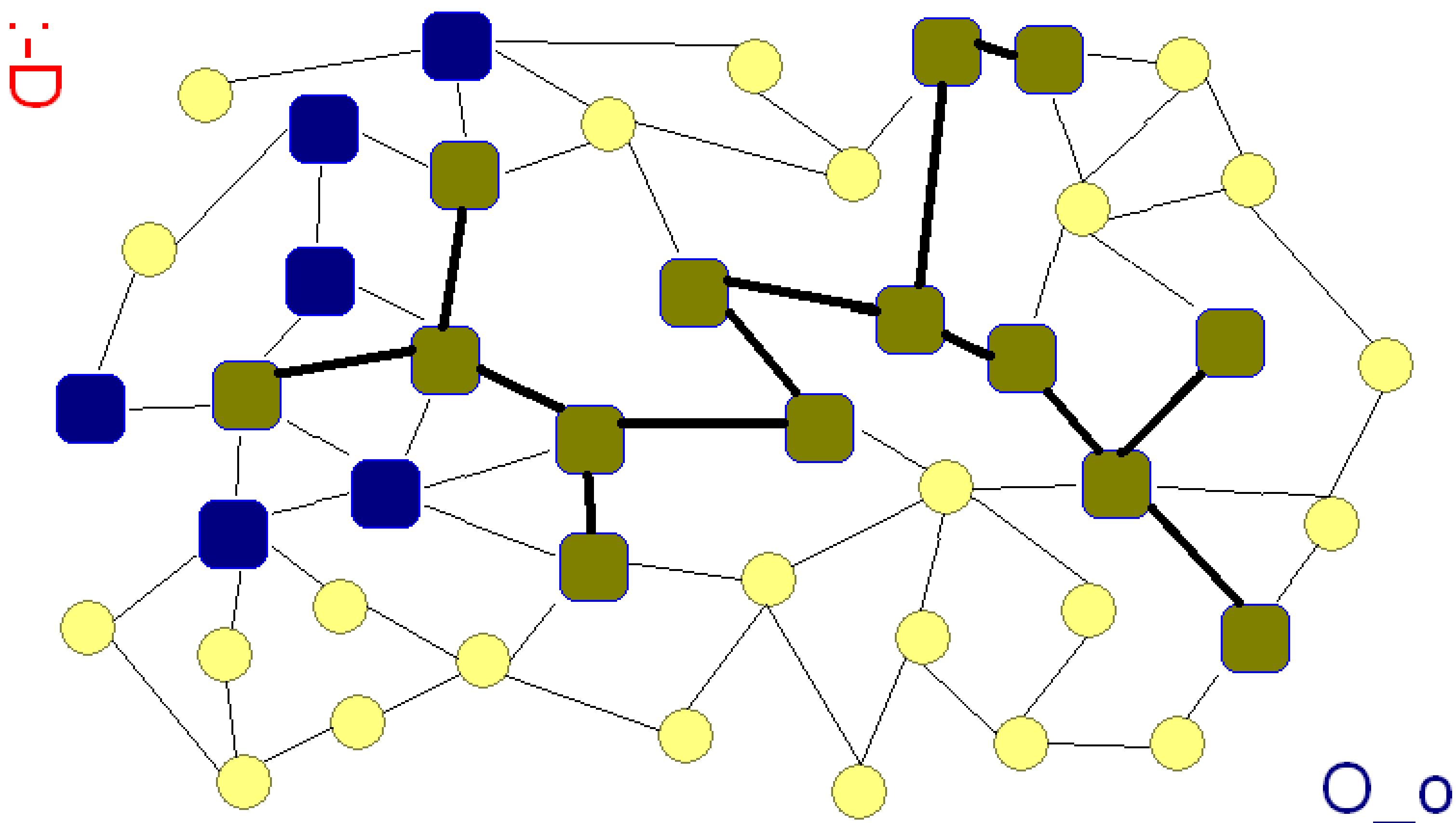
# Applications: IP Traceback



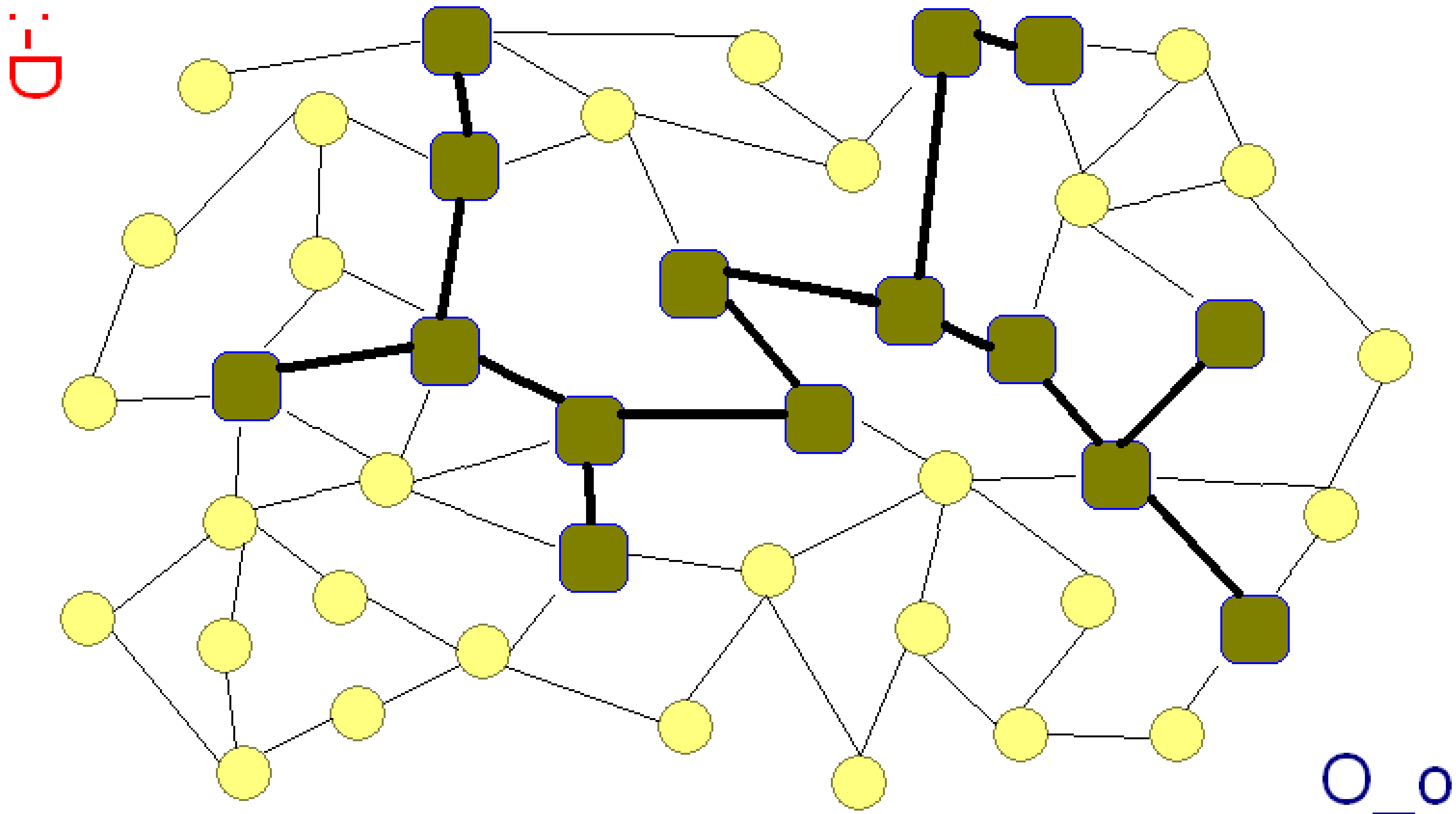
# Applications: IP Traceback



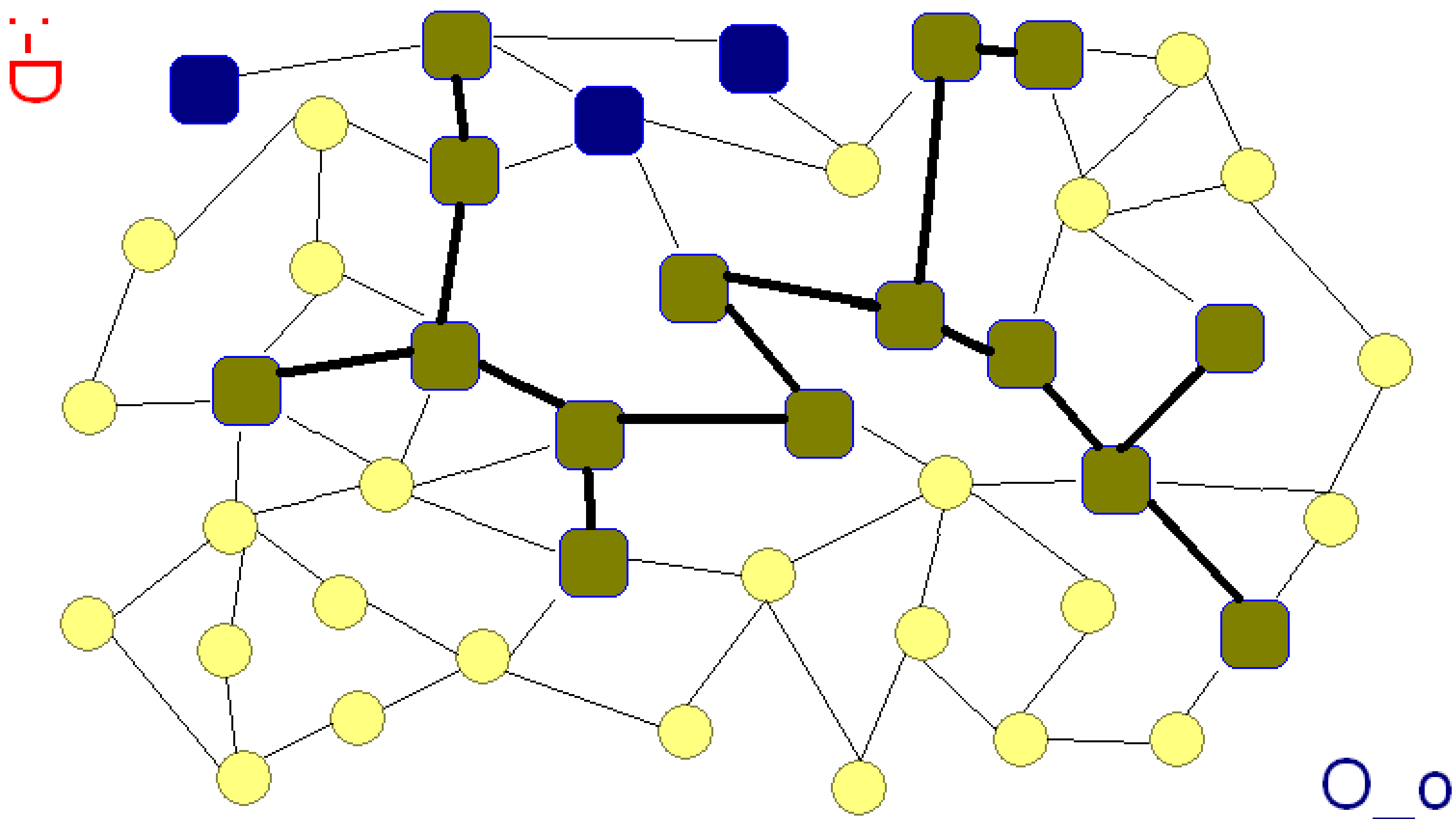
# Applications: IP Traceback



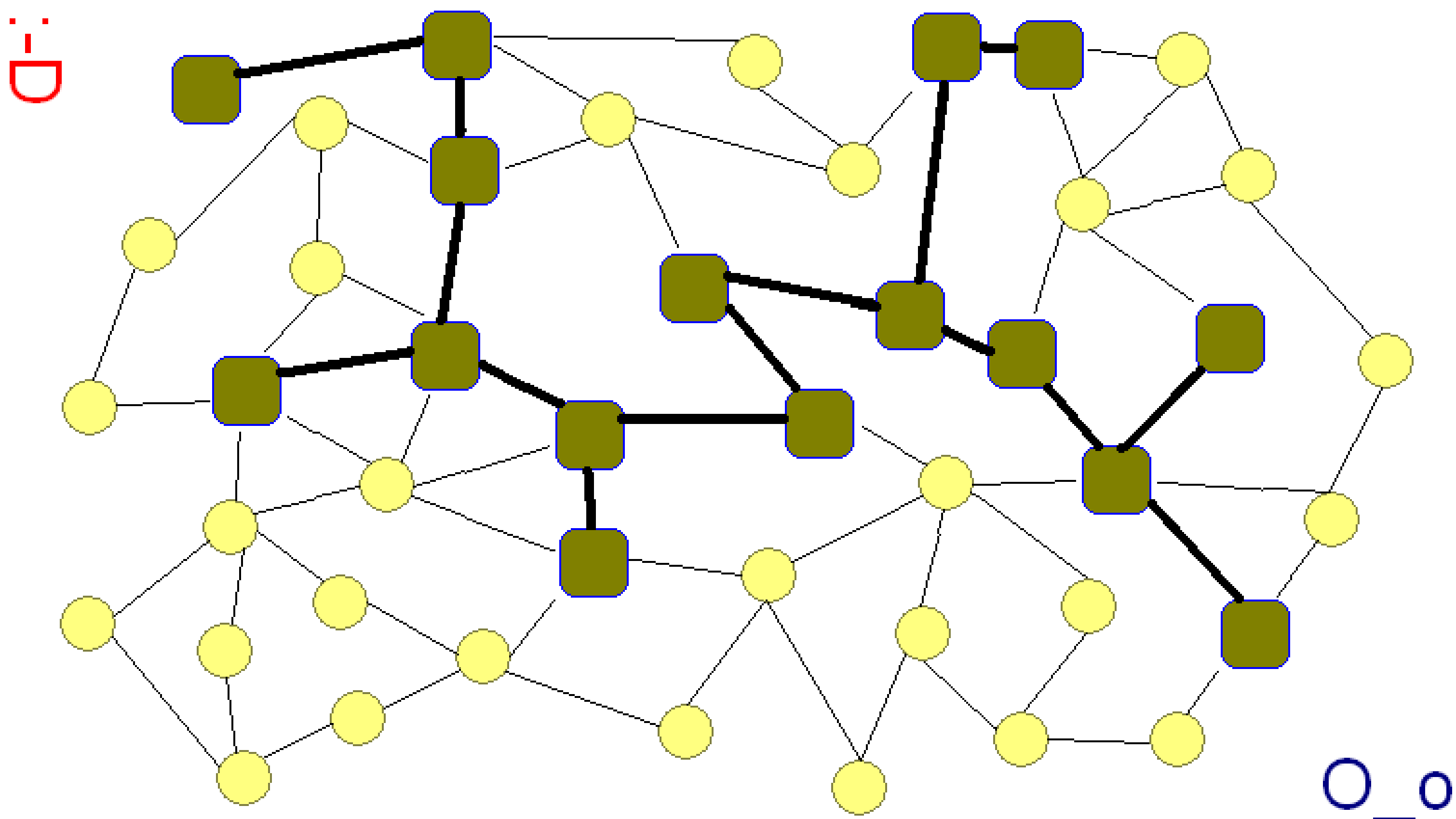
# Applications: IP Traceback



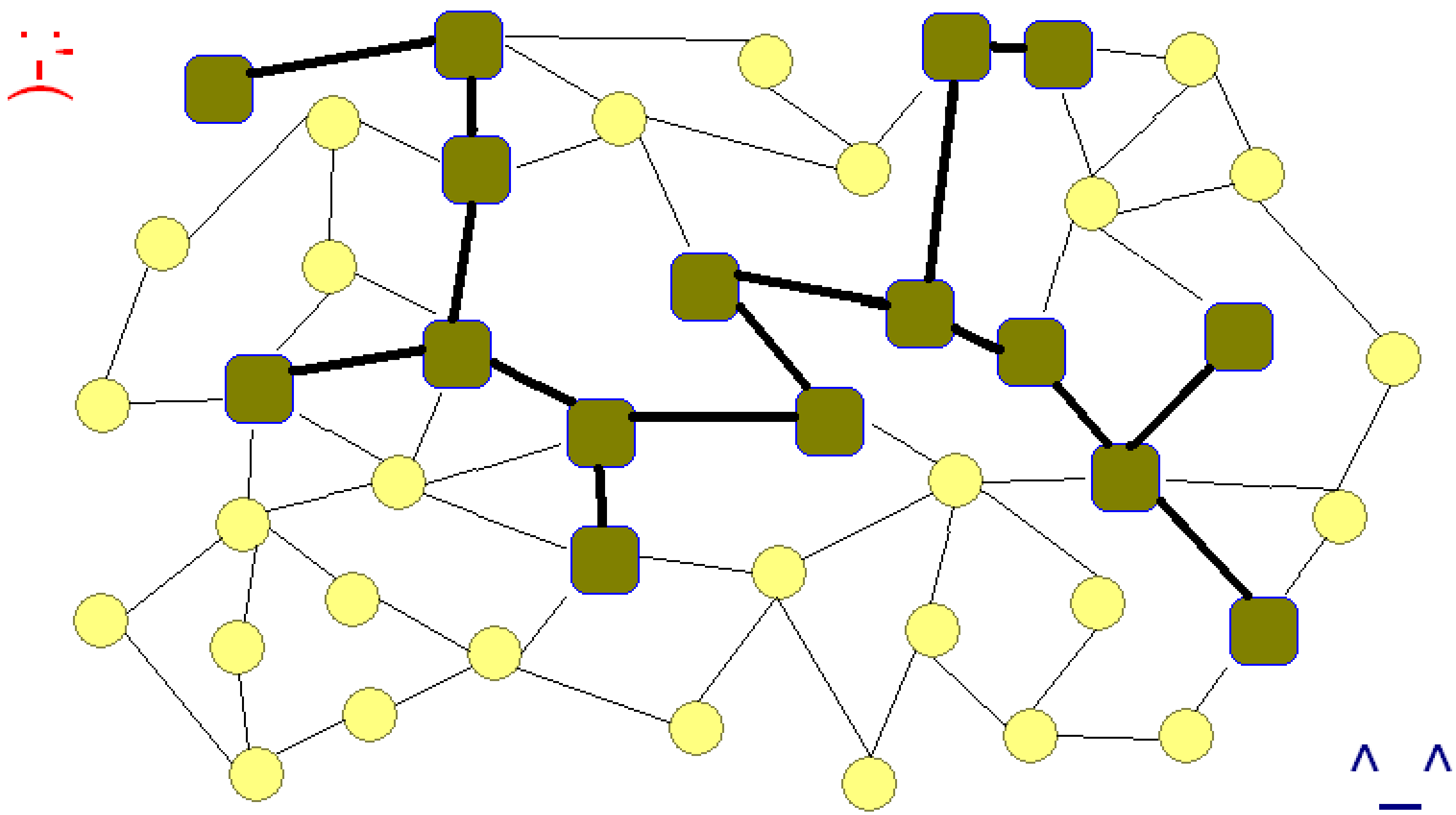
# Applications: IP Traceback



# Applications: IP Traceback



# Applications: IP Traceback



# Applications: Detecting Loops

- Packets sometimes get stuck in loops while traversing the interweb
- Normally packets are labeled with a Time-to-Live field, which is decremented at each hop
- When Time-To-Live is zero, packet is discarded

# Applications: Detecting Loops

- Not a problem caused by well established protocols like TCP/IP
  - Likelihood of a loop occurring is small
- However, experimental protocols may not know if their algorithms are flawed and produce a lot of looping

# Applications: Detecting Loops

- Whitaker and Wetherall propose that for the experimental setting, each packet keep a Bloom Filter of where it's been
- As it passes through the router, the router can check if it is likely that a loop occurred
- Can be made very efficient if each router predetermines its hash and just ORs them into the packets

# Applications: Web Proxy

- A web proxy is a server set up between a network and popular websites
- The proxy is usually "close" to a large user base
- The proxy caches the web content from popular sites

# Applications: Web Proxy

- When you request a web site, the proxy intercepts the request and:
  - 1.) Looks in its cache for the item
  - 2.) Possibly asks other proxies if they have it
  - 3.) Either serves up a local copy, gets a copy from another proxy, or forwards the request to the web site

# Applications: Web Proxy

- The current protocol for web proxies is the Intercache Protocol (ICP):
  - If a cache miss occurs, spam all other proxies to check if they have the missing item
  - Does not scale very well

# Applications: Web Proxy

- Augment the proxies to have Bloom Filters
  - The filters record what files they have
  - Initially, they send each other their filters
- When a cache miss occurs, check all the filters from each proxy for 'likely' candidates
  - Only spam those candidates

# Applications: Web Proxy

- At various intervals, the proxies send updates to each other (as their caches change over time)
- Fan et al showed in a simulated environment that Bloom Filter Proxies:
  - Reduce the number of inter-proxy messages by a factor of 25-60
  - Reduce bandwidth used by proxies by 50%
  - Eliminate over 30% CPU overhead

# Bloom Filters: A Summary

- Bloom Filters are:
  - Easy to implement
  - Fun to use
  - Space efficient beyond belief
  - Useful in many systems applications
- However, must know when to use them

# Bloom Filters: A Summary

| | Memory | Runtime | Error Rate |
|---|---|---|---|
| **Bloom Filter** | O(cn) | O(cm) | $(.6815)^c$ |
| **Hash Set** | O(mn) | $O(m(1 + \alpha))$ | 0 |

# The End… ?

- A parting comment on Bloom Filters by Andrei Broder:
  - Whenever a list or set is used, and space is at a premium, consider using a Bloom Filter if the effects of false-positives can be mitigated

# Bonus Material!

- Turns out there is *another* kind of hash set, also called the hash set
- Was commonly used before Bloom Filters took over
- Takes up slightly more memory, runs slightly faster, has slightly better error rates than a Bloom Filter
  - Very useful in specialized applications

# A Hash Set Implementation

- Same assumptions: **n** elements, each **m** bits long

- Same implementation as the naïve set, except instead of storing the element, store its hash
  - Represent the element using $c * \log_2(n)$ bits, where $c$ is a constant we can choose
    - As will be seen later, $c$ is usually very small

# A Hash Set: Time Analysis

- add: $O(m + c \log^2(n))$
- remove: $O(m + c \log^2(n))$
- contains: $O(m + c \log^2(n))$
- num_occurs: $O(m + c \log^2(n))$
  - "Set" can be converted to a "multiset" by extending each element with a counter

# A Hash Set: Space Analysis

- O(c log(n) n) storage
  - Stores **n** elements, each **c log(n)** bits long
  - Assume counter size is negligible

# Hash Set: Error Analysis

- Great savings all around at no extra penalty, right?
  - Wrong! May result in erroneous behavior

- Query operations may not function correctly:
  - `contains<T>(T item)` may produce wrong answers
  - `num_occurs<T>(T item)` may produce wrong answers

# Hash Set: Error Analysis

- `contains<T>(T item)` may produce wrong answers if a hash collision occurs
- Hash collisions never produce false-negatives
  - *E.g.*, if set is {1, 2, 3, 4, 5}, will never report 5 is not in the set
- Hash collisions may report false-positives
  - *E.g.*, if set is {1, 2, 3, 4, 5}, may say that element 6 is in the set

# Hash Set: Error Analysis

- `num_occurs<T>(T item)` may produce wrong answers if a hash collision occurs
- Hash collisions may never decrease the counter
  - *E.g.,* if the set is {1, 1, 2, 3}, will never say that element 1 occurs once or less
- Hash collisions may increase the counter
  - *E.g.,* if the set is {1, 1, 2, 3}, may say that element 1 occurs three times

# Hash Set: Error Analysis

- Probability of a hash collision:

A

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |

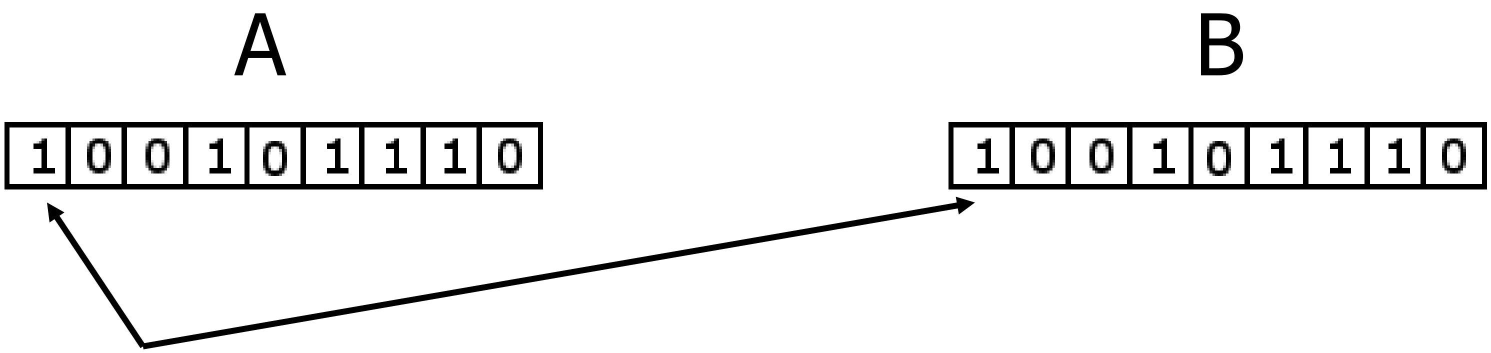
B

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |

- Probability of any two bits being identical is ½ for different objects
  - Follows from uniform mapping property of hash function

# Hash Set: Error Analysis

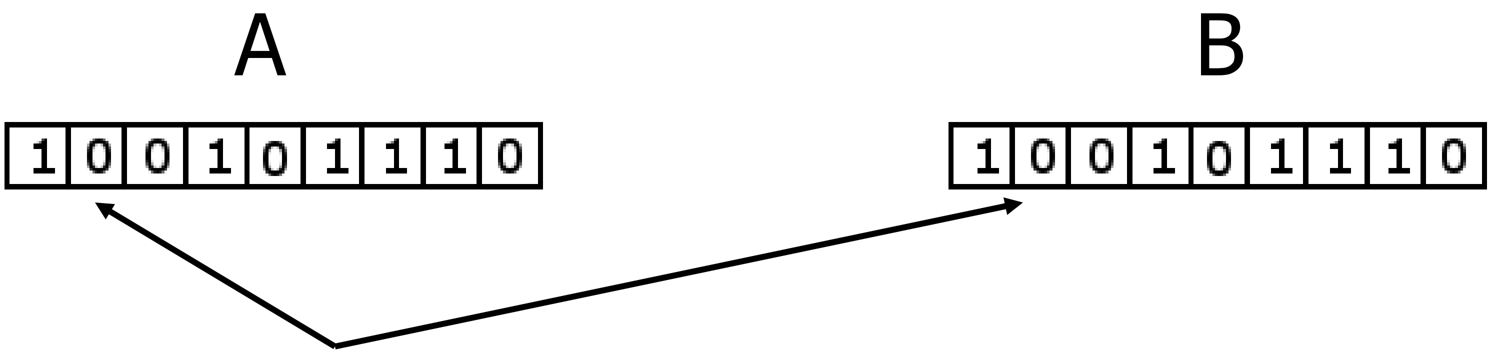
- Probability of a hash collision:




- ½

# Hash Set: Error Analysis

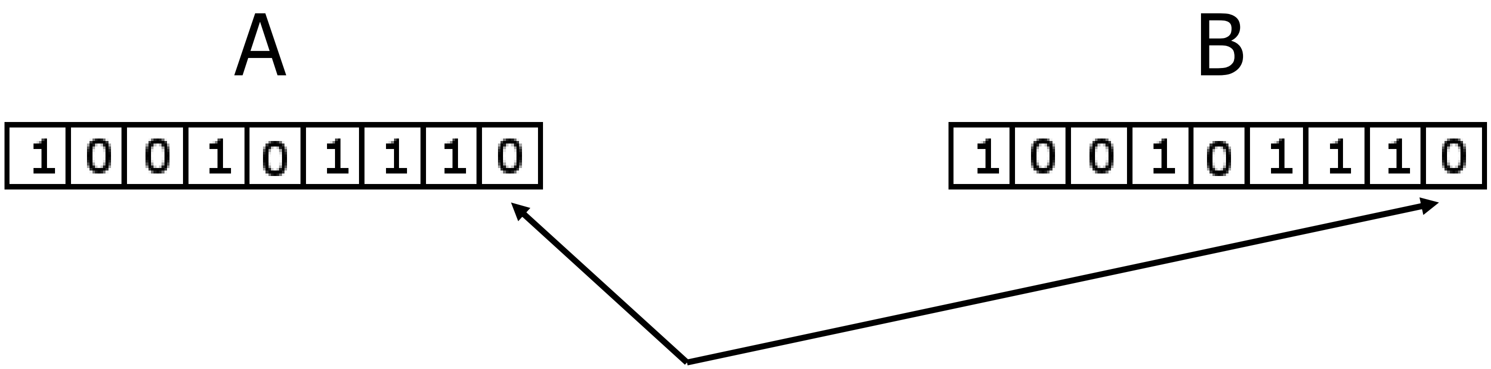
- Probability of a hash collision:




- ½ * ½

# Hash Set: Error Analysis

- Probability of a hash collision:




- $\frac{1}{2} * \frac{1}{2} * \ldots * \frac{1}{2} = \left(\frac{1}{2}\right)^{c \log_2(n)}$

# Hash Set: Error Analysis

- Probability of a hash collision:

A

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |

B

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |

- $\frac{1}{2} * \frac{1}{2} * \ldots * \frac{1}{2} = (\frac{1}{2})^{c \log_2(n)}$
- $(\frac{1}{2})^{c \log_2(n)} = 2^{\log_2(1/n^c)}$

# Hash Set: Error Analysis

- Probability of a hash collision:

| A | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |

| B | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |

- $\frac{1}{2} * \frac{1}{2} * \ldots * \frac{1}{2} = (\frac{1}{2})^{c \log_2(n)}$
- $(\frac{1}{2})^{c \log_2(n)} = 2^{\log_2(1/n^c)}$
- $2^{\log_2(1/n^c)} = 1/n^c$

# Hash Set: Error Analysis

- Since we have **n** elements, a collision can occur with any one of them

- Probability of a collision is at most (by union bound):

$$n * 1/n^c = 1/n^{(c-1)}$$

# Hash Set: Error Analysis

- Probability of `contains<T>` producing a false positive: $1/n^{(c-1)}$
- Probability of `num_occurs<T>` producing an artificially high value for an element that is in the set is bounded above by $1/n^{(c-1)}$
- In practice, c is set to 2 or 3

# Hash Sets: A Summary

| | Memory | Runtime | Error Rate |
|---|---|---|---|
| **Bloom Filter** | $O(cn)$ | $O(cm)$ | $(.6815)^c$ |
| **Hash Set** | $O(c' n \log n)$ | $O(m+c'\log^2 n)$ | $1/n^{c-1}$ |
| **Original Hash Set** | $O(mn)$ | $O(m(1 + \alpha))$ | 0 |