Probability in Computing

LECTURE 6: BINS AND BALLS,
APPLICATIONS: HASHING & BLOOM FILTERS

Agenda

- Review: the problem of bins and ba
- Poisson distribution
- Hashing
- Bloom Filters

Balls into Bins

- We have m balls that are thrown into n bi
with the location of each ball chosen independently and uniformly at random fro possibilities.
- What does the distribution of the balls into bins look like
  - "Birthday paradox" question: is there a bin wit least 2 balls
  - How many of the bins are empty?
  - How many balls are in the fullest bin?

Answers to these questions give solutions to many problems in the design and analysis algorithms

The maximum load

- When n balls are thrown independently and unif random into n bins, the probability that the maxi load is more than 3 ln*n*/lnln*n* is at most 1/*n* for *n* sufficiently large.
  - By Union bound, Pr [bin 1 receives ≥ M balls] ≤ $\binom{n}{M}($
  - Note that:

$$\binom{n}{M}\left(\frac{1}{n}\right)^M \le \frac{1}{M!} \le \left(\frac{e}{M}\right)^M.$$

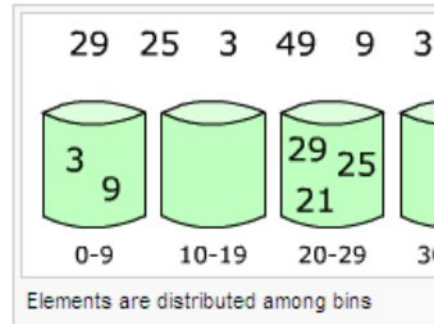
  - Now, using Union bound again, Pr [any ball receives ≥ is at most

$$n\left(\frac{e}{M}\right)^M \le n\left(\frac{e\ln\ln n}{3\ln n}\right)^{3\ln n/\ln\ln n}$$

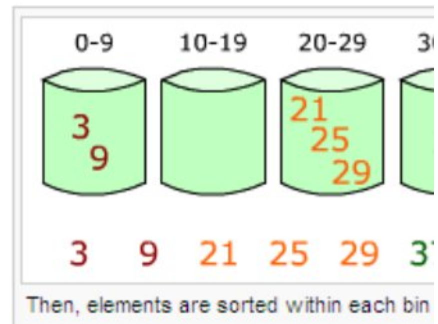
which is ≤ 1 / n

Application: Bucket Sort

- A sorting algorithm that breaks the $\Omega(n\log n)$ lower bound under certain input assumption
- Bucket sort works as follows:
  - Set up an array of initially empty "buckets."
  - Scatter: Go over the original array, putting each object in its bucket.
  - Sort each non-empty bucket.
  - Gather: Visit the buckets in order and put all elements back into the original array.



Elements are distributed among bins



Then, elements are sorted within each bin

- A set of $n = 2^m$ int randomly chosen $[0,2^k), k \geq m$, can be in expected time (
  - Why: will analyz

The Poisson Distribution

- Consider m balls, n bins
  - Pr [a given bin is empty] = $\left(1-\frac{1}{n}\right)^m \approx e^{-m/n};$
  - Let X_j is a indicator r.v. that is 1 if bin j empty, 0 otherw
  - Let X be a r.v. that represents # empty bins

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i] = n\left(1-\frac{1}{n}\right)^m \approx n e^{-m/n}$$

  - Generalizing this argument, Pr [a given bin has r balls] =

$$\binom{m}{r}\left(\frac{1}{n}\right)^r\left(1-\frac{1}{n}\right)^{m-r} = \frac{1}{r!}\frac{m(m-1)\cdots(m-r+1)}{n^r}\left(1-\frac{1}{n}\right)^{m-r}.$$

  - Approximately, $p_r \approx \dfrac{e^{-m/n}(m/n)^r}{r!}$
  - So: **Definition 5.1:** *A discrete* Poisson random variable X *with parameter* *following probability distribution on* $j = 0, 1, 2, \ldots$:

$$\Pr(X = j) = \frac{e^{-\mu}\mu^j}{j!}.$$

Limit of the Binomial Distributio

We have shown that, when throwing m balls randomly into b bins, the probability that a bin has r balls is approximately the Poisson distribution with mean m/b. In g eral, the Poisson distribution is the limit distribution of the binomial distribution w parameters n and p, when n is large and p is small. More precisely, we have the lowing limit result.

Theorem 5.5: *Let X_n be a binomial random variable with parameters n and p, wh p is a function of n and $\lim_{n\to\infty} np = \lambda$ is a constant that is independent of n. Th for any fixed k,*

$$\lim_{n\to\infty} \Pr(X_n = k) = \frac{e^{-\lambda}\lambda^k}{k!}.$$

This theorem directly applies to the balls-and-bins scenario. Consider the situat where there are m balls and b bins, where m is a function of b and $\lim_{n\to\infty} m/b =$ Let X_n be the number of balls in a specific bin. Then X_n is a binomial random varia with parameters m and $1/b$. Theorem 5.5 thus applies and says that

$$\lim_{n\to\infty} \Pr(X_n = r) = \frac{e^{-m/n}(m/n)^r}{r!},$$

Application: Hashing

- The balls-and-bins model is good to model hashi
- Example: password checker
  - Goal: prevent people from choosing common, easily cra passwords
  - Keeping a dictionary of unacceptable passwords and che created password against this dictionary.
- Initial approach: Sorting this dictionary and do bir search on it when checking a password
  - Would require $\Omega(\log m)$ time for m words in the dictiona
- New approach: chain hashing
  - Place the words into bins and search appropriate bin for
  - The worlds in a bin: implemented as a linked list
  - The placement of words into bins is done by using a has

Chain hashing

- Hash table
  - A hash function f: U → [0,n-1] is a way of placing items universe U into n bins
  - Here, U consists of all possible password strings
  - The collection of bins called hash table
  - Chain hashing: items that fall into the same bin are chai together in a linked list
- Using a hash table turns the dictionary problem i balls-and-bins problem
  - m words, hashing range [0..n-1] → m balls, n bins
  - Making assumption: we can design perfect hash functio words into bins uniformly random
    - A given word could be mapped into any bin with the same

Search time in chain hashing

- To search for an item
  - First hash it to find the corresponding bin then it in the bin: sequential search through the link list
  - The expected # balls in a bin is about m/n ➔ expected time for the search is $\Theta(m/n)$
  - If we chose m=n then a search takes expected constant time
- Worst case
  - maximum # balls in a bin: $\Theta({}^{\ln n}/_{\ln\ln n})$ if choose
  - Another disadvantage: wasting a lot of space i empty bins

Hashing: bit strings

- In chain hashing, n balls n bins, we waste a lot empty bins ➔ should have m/n >>1
- Hashing using sort fingerprints will help
  - Suppose: passwords are 8-char, i.e. 64 bits
  - We use a hash function that maps each pwd into a 32-string, i.e. a fingerprint
  - We store the dictionary of fingerprints of the unaccept passwords
  - When checking a password, compute its fingerprint the check it against the dictionary: if found then reject this password
- But it is possible that our password checker may give the correct answer!

False positives

- This hashing scheme gives a false pos when it rejects a good password
  - The fingerprint of this password accidental matches that of an unacceptable passwor
  - For our password checker application this conservative approach is, however, accept the probability of making a false positive is too high

False positive probability

- How many bits should we use to create fingerprints?
  - We want reasonably small probability of a fals
    positive match
  - Prob [the fingerprint of a given good pwd ≠ ar
    unacceptable fingerprint] = 1- $^1/_{2^b}$; here b # b
  - Thus for m unacceptable pwd, prob [false pos
    occurs on a given good pwd] = 1- $(1- {^1/_{2^b}})^m \geq$
  - Easy to see that: to make this prob less than a
    small constant, we need b= $\Omega(\log n)$
    - If use b=2log*m* bits ➔ Prob [a false positive]= 1-(
      $^1/_m$
    - Dictionary of 2^{16} words using 32-bit fingerprint ➔ fa
      $^1/_{65,536}$

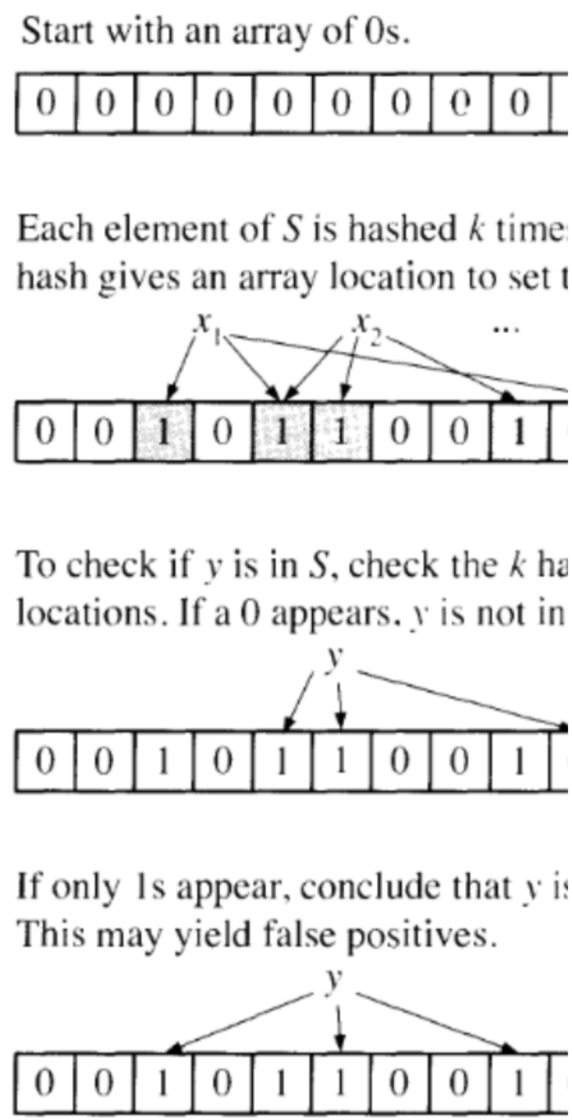
An approximate set membershi problem

- Suppose we have a set $S = \{s_1, s_2, s_3, \ldots s_m\}$ of m elements from a large universe s U. We would like to represent the elemen S in such a way so that
  - We can quickly answer the queries of form "I an element of S?"
  - We want the representation take as little spa possible
- For saving space we can accept occasion mistakes in form of false positives
  - E.g. in our password checker application

Bloom filters

- A Bloom filter: a data structure for this approximate set membership problem
  - By generalizing these mentioned hashing idea achieve more interesting trade-off between required space and the false positive probabil
  - Consists of an array of *n* bits, A[0] to A[n-1], initially set to 0
  - Uses *k* independent hash functions h_1, h_2, ..., with range {0,...n-1}; all these are uniformly random
  - Represent an element $s \in S$ by setting $A[h_i(s)]$ i=1,..k

- Checking: For any value x, to see if $x \in S$ simply check if $A[h_i(x)] = 1$ for all $i=1,..k$
  - If not, clearly x is not a member of S
  - If right, we assume that x is in S but we could be wrong! ➔ false positive



False positive probability

- The probability of a false positive for an element the set
  - After all m elements of S are hashed into Bloom filter, P give bit =0] = $(1-{}^{1}/_{n})^{km} \approx e^{-km/n}$. Let p= $e^{-km/n}$.
  - Prob [a false positive] = $(1- (1-{}^{1}/_{n})^{km})^{k} \approx (1-e^{-km/n})^{k}$ = (Let f= $(1-p)^{k}$.
  - Given m, n what is the optimum k to minimize f?
    - Note that a higher k gives us more chance to find a 0-bit f element not in S, but using fewer h-functions increases th of 0-bit in the array.
  - Optimal k = $\ln 2 . {}^{n}/_{m}$ which reaches minimum f = $½^{k}$ $\approx (0.6185)^{n/m}$
  - Thus Bloom filters allow a small probability of a false po while keep the number of storage bit per item a constar
    - Note in previous consideration of fingerprints we need Ω(l per items

Bloom filters: applications

- Discovering DoS attack attempt
  - Computing the difference between SYN and FIN packets
    - Matching between SYN and FIN packets by tuples of addresses (source and destination po
- Many, many other applications

Application of hashing: breaking symmetry

- Suppose that n users want a unique resource (processes demand CPU time) how can we decide permutation quickly and fairly?
  - Hashing the User ID into 2^b bits then sort the resulting r
    - That is, smallest hash will go first
    - How to avoid two users being hashed to the same value?
- If b large enough we can avoid such collisions as birthday paradox analysis
  - Fix an user. Prob [another user has the same hash] = 1 $^1/_{2^b})^{n-1} \le {}^{(n-1)}/_{2^b}$
  - By union bound, prob [two users have the same hash] =
    - Thus, choosing b =3log*n* guarantees success with probabi
  - Leader election

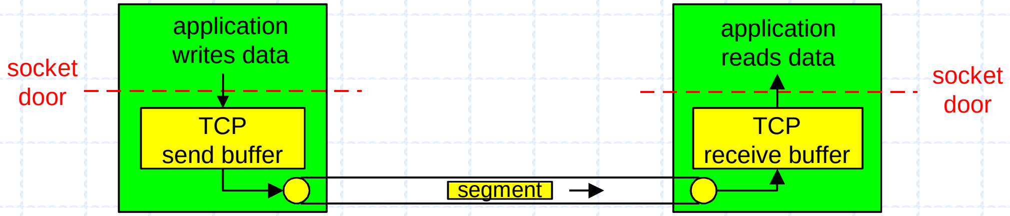
SYN FLOOD DEFENSE SOLUTIONS

TCP SYN-Flooding Attack

- TCP services are often susceptible to vari types of DoS attacks
  - SYN flood: external hosts attempt to overwhe server machine by sending a constant stream connection requests
    - Streaming spoofed TCP SYNs
    - Forcing the server to allocate resources for each new con until all resources are exhausted
  - 90% of DoS attacks use TCP SYN floods
  - Takes advantage of three way handshake
    - Server start “half-open” connections
    - These build up... until queue is full and all additional requ blocked

TCP: Overview RFCs: 793, 1122, 1323, 20 2581

- **point-to-point:**
  - one sender, one receiver
- **reliable, in-order *byte steam:***
  - no “message boundaries”
- **pipelined:**
  - TCP congestion and flow control set window size
- ***send & receive buffers***



- **full duplex data:**
  - bi-directional data
  same connection
  - MSS: maximum seg
  size
- **connection-oriente**
  - handshaking (exch
  control msgs) init’s
  sender, receiver sta
  before data exchar
- **flow controlled:**
  - sender will not ove
  receiver

TCP segment structure

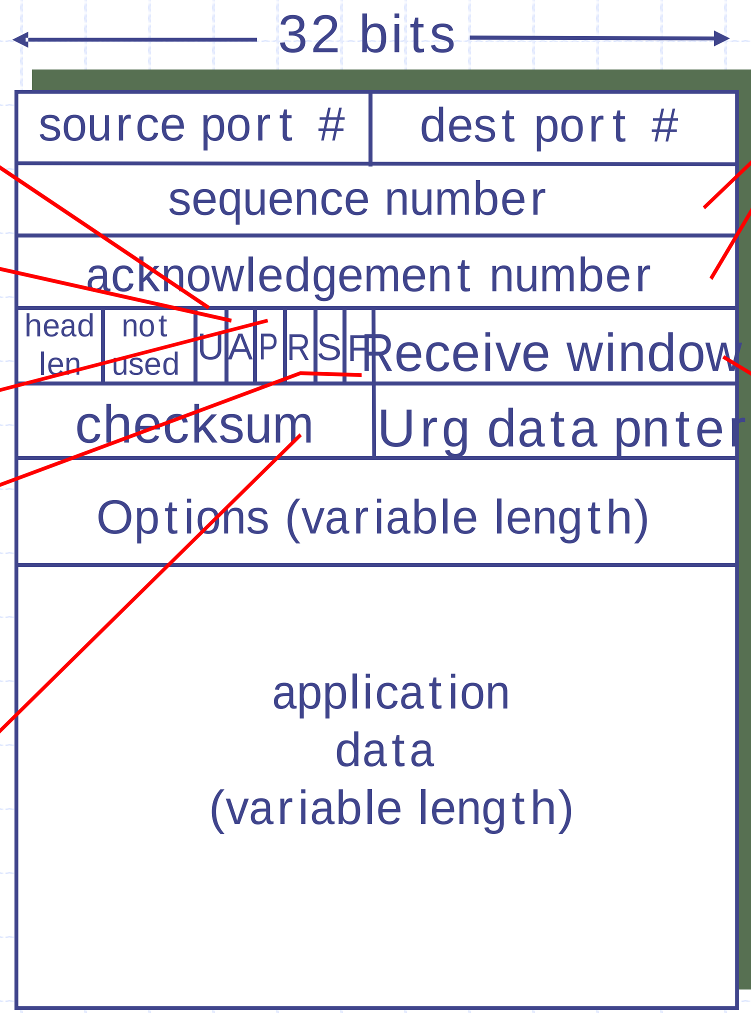
URG: urgent data
(generally not used)

ACK: ACK #
valid

PSH: push data now
(generally not used)

RST, SYN, FIN:
connection estab
(setup, teardown
commands)

Internet
checksum
(as in UDP)

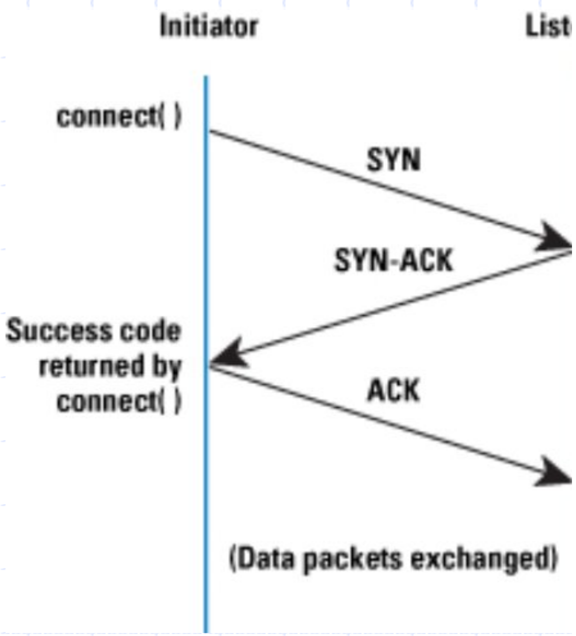


count
by by
of da
(not s

l
rcv
to

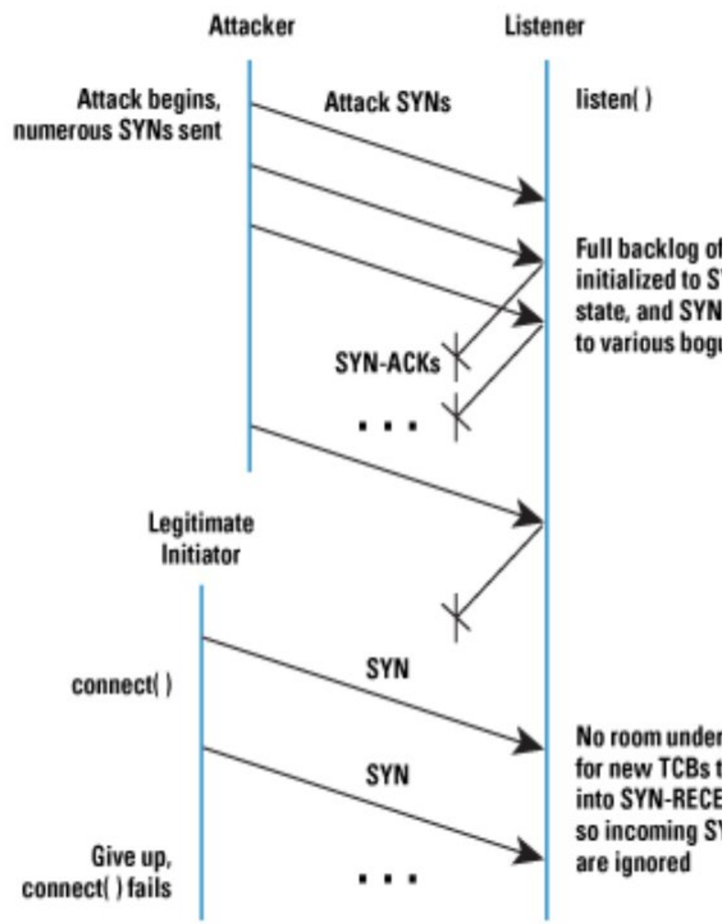
Attack Mechanism

- Transmission Control Block (TCB) is reserved
- TCP SYN-RECEIVED state: connection is half-opened
  - Up on receiving SYN, segment TCB
  - Transited to ESTABLISHED until last ACK

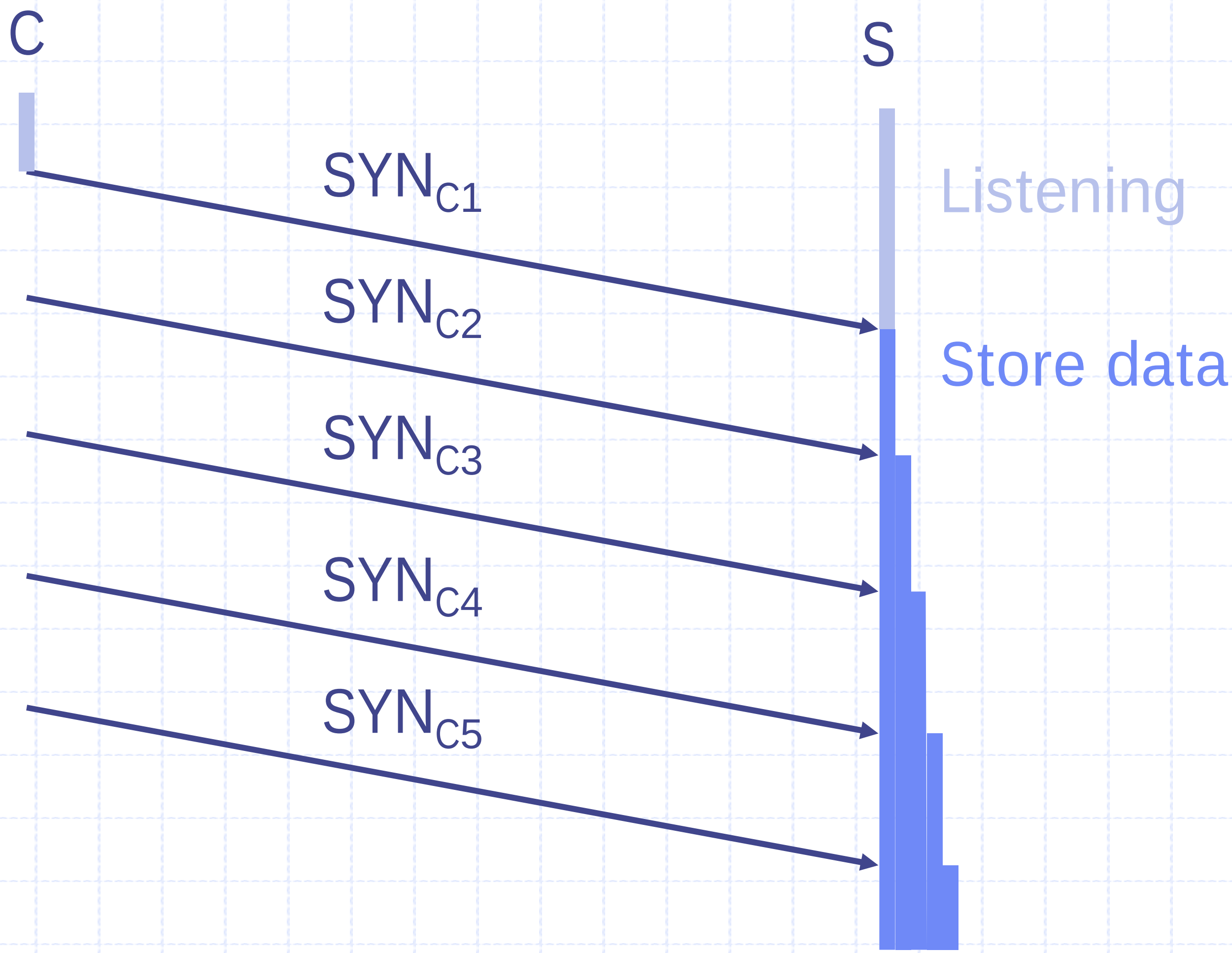


Attack Mechanism

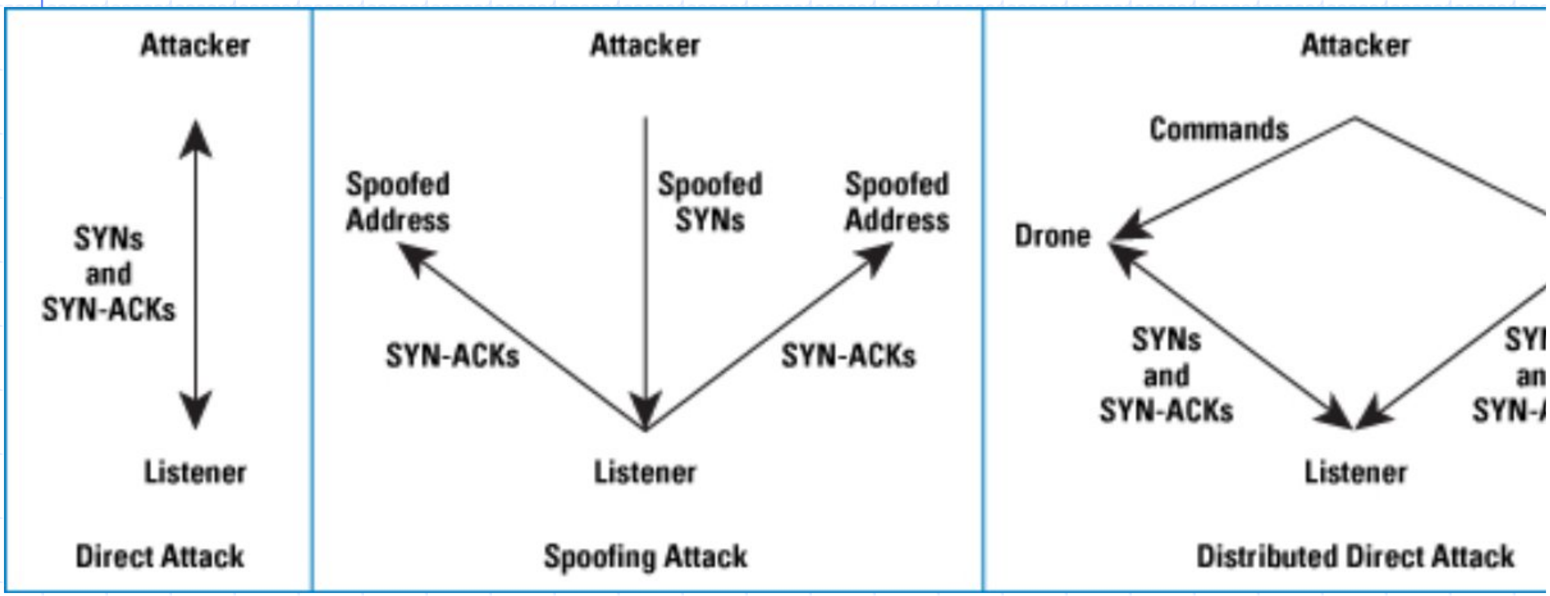
- attacker sends a flood of SYNs ➔ too manyTCB ➔ host is exhauted in memory.
- To avoid this, OS only allows a fixed maximum number of TCBs in SYN-RECEIVED
- If this threshold is reached, new coming SYN will be rejected



SYN Flooding



Implementation Method



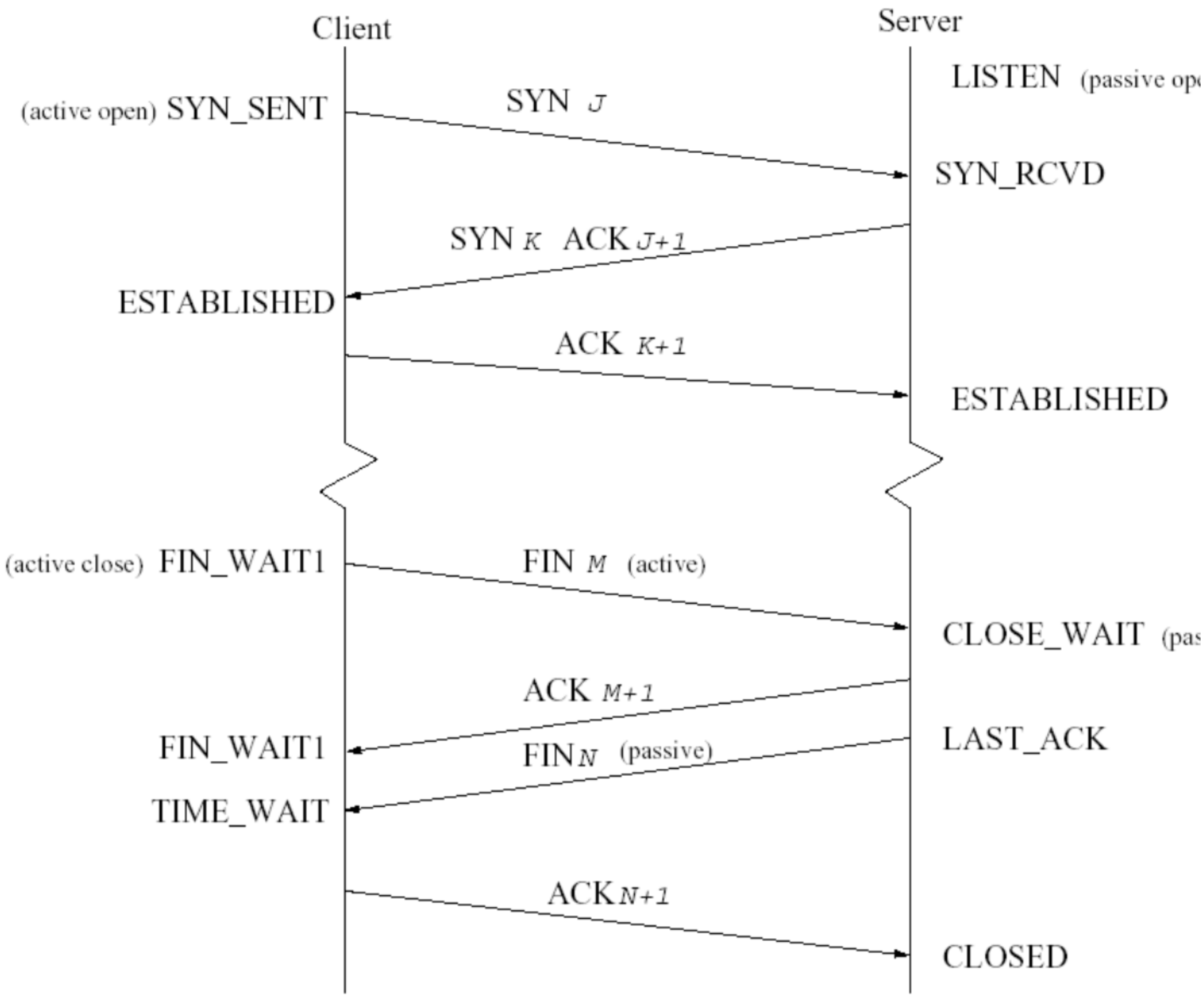
How to create a successful flood

- Making drops of incomplete connection (IC)
  - Standard TCP: a connection times out only after some retranmisstion ▪
  - Assuming 1024 ICs are allowed per socket➔ 2 connection attempts pe exhaust all allocated resources.
  - Note that existing ICs are dropped when a new SYN request is receive
- If an ACK arrives at the server but does not find a correspor state ➔ the server fail to establish such required connectior
  - Round trip time (RTT): time required for the server to have the client r
  - Forcing the server to drop IC state at a rate larger than the RTT, ➔ nc are able to complete ➔ success in attack!
- The goal of attack is to recycle every connection before the RTT
  - For a listen queue size of 1024, and a 100 millisecond RTT ➔ need 10 per second.
  - A minimal size TCP packet is 64 bytes, so the total bandwidth used is c 4Mb/second ➔ practical!

Flood Detection System (FD

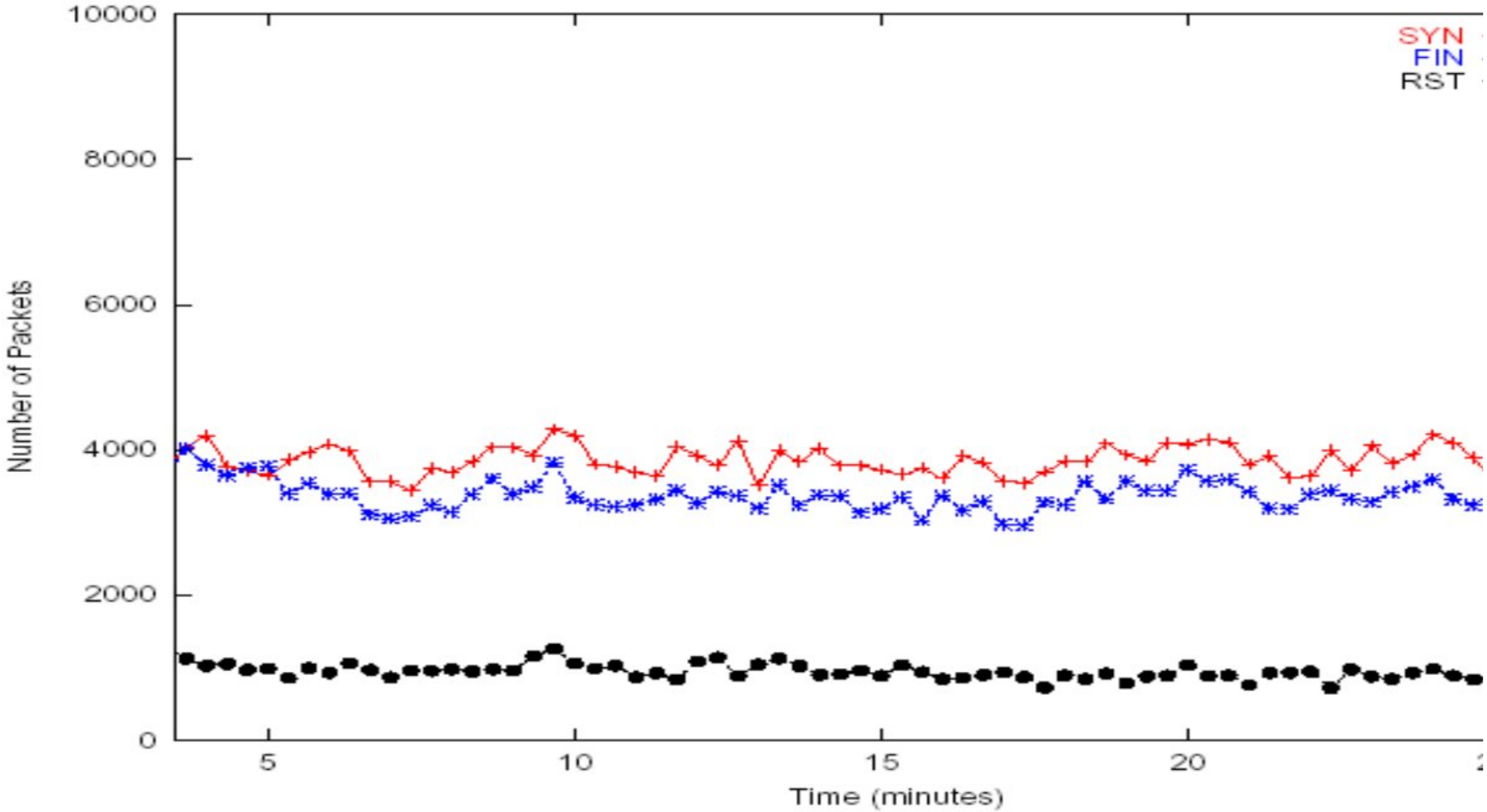
- Stateless, simple, edge (leaf) routers
- Utilize SYN-FIN pair behavior
- Include (SYNACK - FIN) so client or server
- However, RST violates SYN-FIN behav
- Placement: First/last mile leaf routers
  - First mile – detect large DoS attacker
  - Last mile – detect DDoS attacks that first would miss

SYN – FIN Behavior



SYN – FIN Behavior

- Generally every SYN has a FIN



SFD-Method

1- Classification of packets

2-Computing the # of SYN and FIN packe
going through

3-Using algorithm CUSUM to analyze the
(SYN-FIN) pair behaviour

SFD-BF Method

- Improvement on previous SFD:
  - Compute the difference between #SYN an #FIN when the packets are matched on th tuple:
    - When a SYN packet comes, determine the corresponding 4-tuple and insert this into I Increase the counter specified by this 4 tuple.
    - When a FIN/RST packet comes: determ the 4-tuples and find it's hash in BF to decrease the corresponding counter

Intentional Dropping Scheme SYN Flooding Mitigation

Idea

- Normally, if it does not receive a SYN-ACK aft sending a SYN for a certain time a client mac then would resend another SYN until it gets connected to the wanted server.
- The idea of this method is to drop all the first from all the source machine, which would hel reduce SYN flood which is usually first SYNs v spoofed addresses

Method

- The solution is to propose using 3 different B
  - BF1: stores the 4-tuple address of the firs SYN coming from a given source
  - BF2: stores the 4-tupple of all SYNs, with which the 3-way handshake is already completed
  - BF-3: Store the 4-tupple of other SYNs.

Method

Once a SYN arrives, its 4-tuple address is checked ag the 3 BFs, where occurs 1 of the 3 following cases

- 1. Not in any BF➔ This is the first SYN the be dropped, also insert the 4-tuple into BF1
- 2. If found in BF-1➔ this is a second SYN w just move the 4-tuple from BF1 to BF3
- 3. If in BF-2 ➔ Let it go through.
- 4. If in BF-3 ➔ let it goes through with probability p=1/n, where n is the value of corresponding counter in BF-3

Method

When an ACK comes, its 4-tupple address is che against the BFs, which may results in 1 of 3 following cases"

1. Not in any BF ➔ drop the packet
2. If it matches one in BF-2 ➔ let it through
3. If in BF-3 ➔ the connection is completed move the 4-tuple address from BF3 to BF-

Result

- First SYN from any source will be dropp
- The second SYN from the same source w go through
- If this same source continue sending SY the probability that the SYN numbered allowed to go through is 1/n

➔ Thus, the SYN flood caused by an attacki source will be mitigated.