Availability Policies

Chapter 7
Outline

• Goals
• Deadlock
• Denial of service
  • Constraint-based model
  • State-based model
• Networks and flooding
• Amplification attacks
Goals

• Ensure a resource can be accessed in a timely fashion
  • Called “quality of service”
  • “Timely fashion” depends on nature of resource, the goals of using it

• Closely related to safety and liveness
  • Safety: resource does not perform correctly the functions that client is expecting
  • Liveness: resource cannot be accessed
Key Difference

- Mechanisms to support availability in general
  - Lack of availability assumes average case, follows a statistical model
- Mechanisms to support availability as security requirement
  - Lack of availability assumes worst case, adversary deliberately makes resource unavailable
  - Failures are non-random, may not conform to any useful statistical model
Deadlock

• A state in which some set of processes block each waiting for another process in set to take some action
  • Mutual exclusion: resource not shared
  • Hold and wait: process must hold resource and block, waiting other needed resources to become available
  • No preemption: resource being held cannot be released
  • Circular wait: set of entities holding resources such that each process waiting for another process in set to release resources

• Usually not due to an attack
Approaches to Solving Deadlocks

• **Prevention**: prevent 1 of the 4 conditions from holding
  • Do not acquire resources until all needed ones are available
  • When needing a new resource, release all held

• **Avoidance**: ensure process stays in state where deadlock cannot occur
  • *Safe state*: deadlock can not occur
  • *Unsafe state*: may lead to state in which deadlock can occur

• **Detection**: allow deadlocks to occur, but detect and recover
Denial of Service

- Occurs when a group of authorized users of a service make that service unavailable to a (disjoint) group of authorized users for a period of time exceeding a defined maximum waiting time
  - First “group of authorized users” here is group of users with access to service, whether or not the security policy grants them access
  - Often abbreviated “DoS” or “DOS”
- Assumes that, in the absence of other processes, there are enough resources
  - Otherwise problem is not solvable unless more resources created
  - Inadequate resources is another type of problem
Components of DoS Model

• *Waiting time policy*: controls the time between a process requesting a resource and being allocated that resource
  - Denial of service occurs when this waiting time exceeded
  - Amount of time depends on environment, goals

• *User agreement*: establishes constraints that process must meet in order to access resource
  - Here, “user” means a process
  - These ensure a process will receive service within the waiting time
Constraint-Based Model (Yu-Gligor)

• Framed in terms of users accessing a server for some services
• *User agreement*: describes properties that users of servers must meet
• *Finite waiting time policy*: ensures no user is excluded from using resource
User Agreement

• Set of constraints designed to prevent denial of service
• $S_{\text{seq}}$ sequence of all possible invocations of a service
• $U_{\text{seq}}$ set of sequences of all possible invocations by a user
• $U_{li,\text{seq}} \subseteq U_{\text{seq}}$ that user $U_i$ can invoke
  • $C$ set of operations $U_i$ can perform to consume service
  • $P$ set of operations to produce service user $U_i$ consumes
  • $p < c$ means operation $p \in P$ must precede operation $c \in C$
  • $A_i$ set of operations allowed for user $U_i$
  • $R_i$ set of relations between every pair of allowed operations for $U_i$
Example

Mutually exclusive resource

• $C = \{ \text{acquire} \}$
• $P = \{ \text{release} \}$
• For $p_1, p_2$, $A_i = \{ \text{acquire}_i, \text{release}_i \}$ for $i = 1, 2$
• For $p_1, p_2$, $R_i = \{ (\text{acquire}_i < \text{release}_i) \}$ for $i = 1, 2$
Sequences of Operations

• $U_i(k)$ initial subsequence of $U_i$ of length $k$
  • $n_o(U_i(k))$ number of times operation $o$ occurs in $U_i(k)$

• $U_i(k)$ safe if the following 2 conditions hold:
  • if $o \in U_{i,seq}$, then $o \in A_i$; and
    • That is, if $U_i$ executes $o$, it must be an allowed operation for $U_i$
  • for all $k$, if $(o < o') \in R_i$, then $n_o(U_i(k)) \geq n_{o'}(U_i(k))$
    • That is, if one operation precedes another, the first one must occur more times than the second
Resources of Services

• $s \in S_{\text{seq}}$ possible sequence of invocations of services

• $s$ blocks on condition $c$
  • May be waiting for service to become available, or processing some response, etc.

• $o_i^*(c)$ represents operation $o_i$ blocked, waiting for $c$ to become true
  • When execution results, $o_i(c)$ represents operation
  • Note that when $c$ becomes true, $o_i^*(c)$ may not resume immediately
Resources of Services

• $s(0)$ initial subsequence of $s$ up to operation $o_i^*(c)$
• $s(k)$ subsequence of operations between $k$-1$^{st}$, $k^{th}$ time $c$ becomes true after $o_i^*(c)$
• $o_i^*(c) \rightarrow^{s(k)} o_i(c)$: $o_i$ blocks waiting on $c$ at end of $s(0)$, resumes operation at end of $s(k)$
• $S_{seq}$ live if for every $o_i^*(c)$ there is a set of subsequences $s(0)$, ..., $s(k)$ such that it is initial subsequence of some $s \in S_{seq}$ and $o_i^*(c) \rightarrow^{s(k)} o_i(c)$
Example

• Mutually exclusive resource; consider sequence

\[ (\text{acquire}_i, \text{release}_i, \text{acquire}_i, \text{acquire}_i, \text{release}_i) \]

with \( \text{acquire}_i, \text{release}_i \in A_i \), \( (\text{acquire}_i, \text{release}_i) \in R_i; o = \text{acquire}_i, o' = \text{release}_i \)

• \( U_i(1) = (\text{acquire}_i) \Rightarrow n_o(U_i(1)) = 1, n_o'(U_i(1)) = 0 \)

• \( U_i(2) = (\text{acquire}_i, \text{release}_i) \Rightarrow n_o(U_i(2)) = 1, n_o'(U_i(2)) = 1 \)

• \( U_i(3) = (\text{acquire}_i, \text{release}_i, \text{acquire}_i) \Rightarrow n_o(U_i(3)) = 2, n_o'(U_i(3)) = 1 \)

• \( U_i(4) = (\text{acquire}_i, \text{release}_i, \text{acquire}_i, \text{acquire}_i) \Rightarrow \)

\[ n_o(U_i(4)) = 3, n_o'(U_i(4)) = 1 \]

• \( U_i(5) = (\text{acquire}_i, \text{release}_i, \text{acquire}_i, \text{acquire}_i, \text{release}_i) \Rightarrow \)

\[ n_o(U_i(5)) = 3, n_o'(U_i(5)) = 2 \]

• As \( n_o(U_i(k)) \geq n_o'(U_i(k)) \) for \( k = 1, \ldots, 5 \), the sequence is safe


Example (con’t)

• Let $c$ be true whenever resource can be released
  • That is, initially and whenever a $release_i$ operation is performed

• Consider sequence: $(acquire_1, acquire_2^*(c), release_1, release_2, \ldots, acquire_k, acquire_{k+1}(c), release_k, release_{k+1}, \ldots)$

• For all $k \geq 1$, $acquire_i^*(c) \xrightarrow{s(1)} acquire_{k+1}(c)$, so this is live sequence
  • Here, $acquire_{k+1}(c)$ occurs between $release_k$ and $release_{k+1}$
Expressing User Agreements

• Use temporal logics

• Symbols
  • □: henceforth (the predicate is true and will remain true)
  • ◇: eventually (the predicate is either true now, or will become true in the future)
  • ⊢: will lead to (if the first part is true, the second part will eventually become true); so A ⊢ B is shorthand for A ⇒ ◇B
Example

• Acquiring and releasing mutually exclusive resource type
• User agreement: once a process is blocked on an acquire operation, enough release operations will release enough resources of that type to allow blocked process to proceed

service resource_allocator

User agreement

\[ \text{in}(acquire) \Rightarrow ((\square \Diamond (\#active\_release > 0) \lor (\text{free} \geq \text{acquire.n})) \]

• When a process issues an acquire request, at some later time at least 1 release operation occurs, and enough resources will be freed for the requesting process to acquire the needed resources
Finite Waiting Time Policy

• *Fairness policy*: prevents starvation; ensures process using a resource will not block indefinitely if given the opportunity to progress

• *Simultaneity policy*: ensures progress; provides opportunities process needs to use resource

• *User agreement*: see earlier

• If these three hold, no process will wait an indefinite time before accessing and using the resource
Example

• Continuing example ... these and above user agreement ensure no indefinite blocking

**sharing policies**

**fairness**

$$\text{at}(acquire) \land \Box \Diamond ((\text{free} \geq \text{acquire.n}) \land (#\text{active} = 0)) \leadsto \text{after}(acquire)$$

$$\text{at}(release) \land \Box \Diamond (#\text{active} = 0) \leadsto \text{after}(release)$$

**simultaneity**

$$\text{in}(acquire) \land (\Box \Diamond (\text{free} \geq \text{acquire.n})) \land (\Box \Diamond (#\text{active} = 0))) \leadsto$$

$$((\text{free} \geq \text{acquire.n}) \land (#\text{active} = 0))$$

$$\text{in}(release) \land \Box \Diamond (#\text{active}_\text{release} > 0)) \leadsto (\text{free} \geq \text{acquire.n})$$
Service Specification

- Interface operations
- Private operations not available outside service
- Resource constraints
- Concurrency constraints
- Finite waiting time policy
Example:

- Interface operations of the resource allocation/deallocation example

**Interface operations**

`acquire(n: units)`

- **exception conditions**: `quota[id] < own[id] + n`
- **effects**: `free' = free - n`
  
  `own[id]' = own[id] + n`

`release(n: units)`

- **exception conditions**: `n > own[id]`
- **effects**: `free' = free + n`
  
  `own[id]' = own[id] - n`
Example (con’t)

• Resource constrains of the resource allocation/deallocation example

resource constraints

1. $\square((free \geq 0) \land (free \leq size))$
2. $(\forall id) \left[ \square(own[id] \geq 0) \land (own[id] \leq quota[id]) \right]$
3. $(free = N) \Rightarrow ((free = N) \text{ UNTIL } (after(acquire) \lor after(release))))$
4. $(\forall id) \left[ (own[id] = M) \Rightarrow ((own[id] = M) \text{ UNTIL } (after(acquire) \lor after(release)))) \right]$
Example (con’t)

• Concurrency constraints of the resource allocation/deallocation example

concurrency constraints

1. □(#active ≤ 1)
2. (#active = 1) ~ (#active = 1)
Denial of Service

• Service specification policies, user agreements prevent denial of service *if enforced*

• These do *not* prevent a long wait time; they simply ensure the wait time is finite
State-Based Model (Millen)

• Unlike constraint-based model, allows a maximum waiting time to be specified
• Based on resource allocation system, denial of service base that enforces its policies
Resource Allocation System Model

• \( R \) set of resource types
• For each \( r \in R \), number of resource units (capacity, \( c(r) \)) is constant; a process can hold a unit for a maximum holding time \( m(r) \)
• \( P \) set of processes
• For each \( p \in P \), state is *running* or *sleeping*
  • When allocated a resource, process is running
  • Multiple process can be in running state simultaneously
  • Each \( p \) has upper bound it can be in running state before being interrupted, if only by CPU quantum \( q \)
  • Example: if CPU considered a resource, \( m(CPU) = q \)
Allocation Matrix

• Rows represent processes; columns represent resources
  • $A: P \times R \rightarrow \mathbb{N}$ is matrix
  • For $p \in P, r \in R$, $A_p(r)$ is number of resource units of type $r$ acquired by $p$
  • As at most $c(r)$ of resource type $r$ exist, at most that many can be allocated at any time

R1: The system cannot allocate more instances of a resource type than it has:

$$(\forall r \in R)[\sum_{p \in P} A_p(r) \leq c(r)]$$
More About Resources

• $T: P \rightarrow \mathbb{N}$ is system time when resource assignment was last changed
  • Think of it as a time vector, each element belonging to one process

• $Q^S: P \times R \rightarrow \mathbb{N}$ is matrix of required resources for each process, not including the resources it already holds
  • So $Q^S_p(r)$ means the number of units of resource type $r$ that process $p$ may need to complete

• $Q^T: P \times R \rightarrow \mathbb{N}$ is matrix of how much longer each process $p$ needs the units of resource $r$

• Predicates $\text{running}(p)$ true if $p$ is in running state; $\text{asleep}(p)$ true otherwise

R2: A currently running process must not require additional resources to run

$\text{running}(p) \implies (\forall r \in R)[Q^S_p(r) = 0]$
States, State Transitions

• Current state of system is \((A, T, Q^S, Q^T)\)
• State transition \((A, T, Q^S, Q^T) \rightarrow (A', T', Q'^S, Q'^T)\)
  • We only care about transitions due to allocation, deallocation of resources
• Three relevant types of transitions
  • Deactivation transition: \(\text{running}(p) \rightarrow \text{asleep}'(p)\); process stops execution
  • Activation transition: \(\text{asleep}(p) \rightarrow \text{running}'(p)\); process starts or resumes execution
  • Reallocation transition: transition in which \(p\) has resource allocation changed; can only occur when \(\text{asleep}(p)\)
Constraints

R3: Resource allocation does not affect allocations of a running process:

\[(\text{running}(p) \land \text{running}'(p)) \Rightarrow (A_p' = A_p)\]

R4: \(T(p)\) changes only when resource allocation of \(p\) changes:

\[(A_p'(CPU) = A_p(CPU)) \Rightarrow (T'(p) = T(p))\]

R5: Updates in time vector increase value of element being updated:

\[(A_p'(CPU) \neq A_p(CPU)) \Rightarrow (T'(p) > T(p))\]
Constraints

R6: When $p$ reallocated resources, allocation matrix updated before $p$ resumes execution:

$$asleep(p) \Rightarrow Q^S_p' = Q^S_p + A_p - A_p'$$

R7: When a process is not running, the time it needs resources does not change:

$$asleep(p) \Rightarrow Q^T_p' = Q^T_p$$

R8: when a process ceases to execute, the only resource it must surrender is the CPU:

$$(running(p) \land asleep'(p)) \Rightarrow A_p'(r) = A_p(r) - 1 \quad \text{if } r = \text{CPU}$$

$$(running(p) \land asleep'(p)) \Rightarrow A_p'(r) = A_p(r) \quad \text{otherwise}$$
Resource Allocation System

• A system in a state \((A, T, Q^S, Q^T)\) such that:
  • State satisfies constraints R1, R2
  • All state transitions constrained to meet R3-R8
Denial of Service Protection Base (DPB)

• A mechanism that is tamperproof, cannot be prevented from operating, and guarantees authorized access to resources it controls

• Four parts:
  • Resource allocation system (see earlier)
  • Resource monitor
  • Waiting time policy
  • User agreement (see earlier; constraints apply to changes in allocation when process transitions from \textit{running}(p) to \textit{asleep}(p)
Resource Monitor

• Controls allocation, deallocation of resources and the timing

• $Q^S_p$ is feasible if $(\forall i)[Q^S_p(r_i) + A_p(r_i) \leq c(r_i)] \land Q^S_p(CPU) \leq 1$
  • If the total number of resources it will be allocated will always be no more than the capacity of that resource, and no more than 1 CPU is requested

• $T_p$ is feasible if $(\forall i)[T_p(r_i) \leq max(r_i)]$
  • Here, $max(r_i)$ max time a process must wait for its needed allocation of units of resource type $i$
Waiting Time Policy

• Let $\sigma = (A, T, Q^S, Q^T)$

• Example finite waiting time policy:

$$(\forall p, \sigma)(\exists \sigma')[\text{running}'(p) \land (T'(p) \geq T(p))]$$

• For every process and state, there is a future state in which $p$ is executing and has been allocated resources

• Example maximum waiting time policy:

$$(\exists M)(\forall p, \sigma)(\exists \sigma')[\text{running}'(p) \land (0 < T'(p) - T(p) \leq M)]$$

• There is an upper bound $M$ to how long it takes every process to reach a future state in which it is executing and has been allocated resources
Two Additional Constraints

In addition to all these, a DPB must satisfy these constraints:

1. Each process satisfying user agreement constraints will progress in a way that satisfies the waiting time policy

2. No resource other than the CPU is deallocated from a process unless that resource is no longer needed

\[(\forall i)[r_i \neq \text{CPU} \land A_p(r_i) \neq 0 \land A_p'(r_i) = 0] \Rightarrow Q^T_p(r_i) = 0\]
Example: DPB

• Assume system has 1 CPU
• Assume maximum waiting time policy in place
• 3 parts to user agreement:
  • $Q^s_p, T_p$ are feasible
  • Process in running state executes for a minimum amount of time before it transitions to a non-running state
  • If process requires resource type, and enters a non-running state, the time it needs the resource for is decreased by the amount of time it was in the previous running state; that is,

$$Q^T_p \neq 0 \land \text{running}(p) \land \text{asleep}'(p) \Rightarrow (\forall r \in R)[Q^T_p(r) \leq \max(0, \max_r Q^T_p(r) - (T'(p) - T(p)))$$
Example: System

- $n$ processes, round robin scheduler with quantum $q$
- Initially no process has any resources
- Resource monitor selects process $p$ to give resources to
  - $p$ executes until $Q^T_p = 0$ or monitor concludes $Q^S_p$ or $T_p$ is not feasible
- Goal: show there will be no denial of service in this system because
  a) no resource $r_i$ is deallocated from $p$ for which $Q^S_p$ is feasible until $Q^T_p = 0$; and
  b) there is a maximum time for each round robin cycle
Claim (a)

• Before $p$ selected, no process has any resources allocated to it
  • So next process with $Q^S_p$ and $T_p$ feasible is selected
  • It runs until it enters the *asleep* state or $q$, whichever is shorter
  • If in *asleep* state, process is done
  • If $q$, monitor gives $p$ another quantum of running time; this repeats until $Q^T_p = 0$, and then $p$ needs no more resources

• Let $m(r)$ be maximum time any process will hold resources of type $r$
  • Let $M(r) = \max_r m(r)$

• As $Q^S_p$ and $T_p$ feasible, $M$ upper bound for all elements of $Q^T_p$
  • $d = \min(q$, minimum time before $p$ transitions to *asleep* state); exists because a process in running state executes for a minimum amount of time before it transitions to a non-running state
Claim (a) (con’t)

• As $Q^S_p$ and $T_p$ feasible, $M$ upper bound for all elements of $Q^T_p$
• $d = \min(q, \text{minimum time before } p \text{ transitions to asleep state})$
  • Exists because a process in running state executes for a minimum amount of time before it transitions to a non-running state
• At end of each quantum, $m'(r) = m(r) - d$
  • By third part of user agreement
• So after floor($M/d + 1$) quanta, $Q^T_p = 0$
  • So no resources deallocated until $(\forall i) Q^T_p(r_i) = 0$
Claim (b)

• $t_a$ is time between resource monitor beginning cycle and when it has allocated required resources to $p$

• Resource monitor then allocates CPU resource to $p$; call this time $t_{CPU}$
  • Done between each quantum

• When $p$ completes, all its resources deallocated; this takes time $t_d$

• As $Q^S_p$ and $T_p$ feasible, time needed to run $p$, including time to deallocate all resources, is:
  $$t_a + \text{floor}(M/d + 1)(q + t_{CPU}) + t_d$$

• So for $n$ processes, maximum time cycle will take is $n$ times this

• Thus, there is a maximum time for each round robin cycle
Availability and Network Flooding

• Access over Internet must be unimpeded
  • Context: flooding attacks, in which attackers try to overwhelm system resources

• If many sources flood a target, it’s a *distributed denial of service attack*
TCP 3-Way Handshake and Availability

- Normal three-way handshake to initiate connection
- Suppose source never sends third message (the last ACK)
  - Destination holds information about pending connection for a period of time before the space is released
Analysis

• Consumption of bandwidth
  • If flooding overwhelms capacity of physical network medium, SYNs from legitimate handshake attempts may not be able to reach the target

• Absorption of resources on destination host
  • Flooding fills up memory space for pending connections, causing SYNs from legitimate handshake attempts to be discarded

• In terms of the models:
  • Waiting time is the time that destination waits for ACK from source
  • Fairness policy must assure host waiting for ACK (resource) will receive (acquire) it
Analysis in Terms of Model

- Waiting time is the time that destination waits for ACK from source
- Fairness policy must assure host waiting for ACK (resource) will receive (acquire) it
  - But goal of attack is to make sure it never arrives
- Yu-Gligor model: finite wait time does not hold
  - So model says denial of service can occur
- Millen model: $T_p(\text{ACK}) > \max(\text{ACK})$
  - $\max(\text{ACK})$ is the time-out period for pending connections
  - So model says denial of service can occur
Countermeasures

• Focus on ensuring resources needed for legitimate handshakes to complete are available
  • So every legitimate client gets access to server

• First approach: manipulate opening of connection at end point
  • If focus is to ensure connection attempts will succeed at some time, focus is really on waiting time
  • Otherwise, focus is on user agreement

• Second approach: control which packets, or rate at which packets, sent to destination
  • Focus is on implicit user agreements
Intermediate Systems

• Approach is to reduce consumption of resources on destination by diverting or eliminating illegitimate traffic so only legitimate traffic reaches destination
  • Done at infrastructure level

• Example: Cisco routers try to establish connection with source (TCP intercept mode)
  • On success, router does same with intended destination, merges the two
  • On failure, short time-out protects router resources and target never sees flood
Track Connection Status

• Use network monitor to track status of handshake

• Example: *synkill* monitors traffic on network
  • Classifies IP addresses as not flooding (good), flooding (bad), unknown (new)
  • Checks IP address of SYN
    • If good, packet ignored
    • If bad, send RST to destination; ends handshake, releasing resources
    • If new, look for ACK or RST from same source; if seen, change to good; if not seen, change to bad
  • Periodically discard stale good addresses
Intermediate Systems near Sources

- D-WARD relies on routers close to the sources to block attack
  - Reduces congestion in network without interfering with legitimate traffic
- Placed at gateways of possible sources to examine packets leaving (internal) network and going to Internet
- Deployed on systems in research lab for 4 months
  - First month: large number of false alerts
  - Tuning D-WARD parameters reduced this number
D-WARD: Observation Component

• Has set of legitimate internal addresses
• Gathers statistics on packets leaving network, discarding packets without legitimate addresses
• Tracks number of simultaneous connections to each remote destination
  • Unusually large number may indicate attack from this network
• Examines connections with large amount of outgoing traffic but little incoming (response) traffic
  • May indicate destination host is overwhelmed
D-WARD: Observation Component

• Also aggregates traffic statistics to each remote address
• Classifies flows as *attack*, *suspicious*, *normal*
  • *Normal*: statistics match legitimate traffic model
  • *Attack*: if not
• Once traffic classified as attack begins to match legitimate traffic model, indicates attack has ended, so flow reclassified as *suspicious*
  • If it stays suspicious for predetermined time, reclassified as *normal*
D-WARD: Rate-Limiting Component

• When attack detected, this component limits amount of packets that can be sent
• This reduces volume of traffic going from this network to destination
• How it limits rate is based on D-WARD’s best guess of amount of traffic destination can handle
  • When flow reclassified as normal, D-WARD raises rate limit until sending rate is as before
D-WARD: Traffic-Policing Component

- Component obtains information from other 2 components
- Based on this, decides whether to drop packets
  - Packets for normal connections always forwarded
  - Packets for other flows may be forwarded provided doing so does not exceed rate limit associated with flow
Endpoint Protection

• Control how TCP state is stored
  • When SYN received, entry in queue of pending connections created
    • Remains until an ACK received or time-out
    • In first case, entry moved to different queue
    • In second case, entry made available for next SYN
  • In SYN flood, queue is always full
    • So, assure legitimate connections space in queue to some level of probability
    • Two approaches: SYN cookies or adaptive time-outs
SYN Cache

• Space allocated for each pending connection
  • But much less than for a full connection

• How it works on FreeBSD
  • On initialization, hash table (syncache) created
  • When SYN packet arrives, system generates hash from header and uses that to determine which bucket to store enough information to be able to send SYN/ACK on the pending connection (and does so)
    • If bucket full, oldest element dropped
  • If peer returns ACK, entry removed and connection created
  • If peer returns RST, entry removed
  • If no response, repeat fixed number of times; if no responses, remove entry
SYN Cookies

• Source keeps state

• How it works
  • When SYN arrives, generate number (syncookie) from header data and random data; use as ACK sequence number in SYN/ACK packet
    • Random data changes periodically
  • When reply ACK arrives, recompute syncookie from information in header

• FreeBSD uses this technique when pending connection cannot be inserted into syncache
Adaptive Time-Out

- Change time-out time as space available for pending connections decreases
- Example: modified SunOS kernel
  - Time-out period shortened from 75 to 15 sec
  - Formula for queueing pending connections changed:
    - Process allows up to $b$ pending connections on port
    - $a$ number of completed connections but awaiting process
    - $p$ total number of pending connections
    - $c$ tunable parameter
    - Whenever $a + p > cb$, drop current SYN message
Other Flooding Attacks

• These use *reflectors* (typically, infrastructure systems) to augment traffic, creating flooding
  • Attacker need only send small amount of traffic; reflectors create the rest
  • Called *amplification attack*

• Hides origin of attack, which appears to come from reflectors
Smurf Attack

• Relies on router forwarding ICMP packets to all hosts on network
• Attacker sends ICMP packet to router with destination address set to broadcast address of network
• Router sends copy of packet to each host on network
  • If attacker sends steady stream of packets, has the effect of sending that stream to all hosts on network
• Example of an amplification attack
DNS Amplification Attack

- Uses DNS resolvers that are configured to accept queries from any host rather than only hosts on their own network
- Attacker sends packet with source address set to that of target
  - Packet has query that causes DNS resolver to send large amount of information to target
  - Example: zone transfer query is a small query, but typically sends large amount of data to target, typically in multiple packets, each larger than a query packet
Pulse Denial of Service Attack

• Like flooding, but packets sent in pulses
  • May only degrade target’s performance, but that may be enough of a denial of service

• Induces 3 anomalies in traffic to target
  • Ratio of incoming TCP packets to outgoing ACKs increases dramatically
    • Rate of incoming packets much higher than system can send ACKs
  • When attacker reduces number of packets to target, number of ACKS drop
  • Distribution of incoming packet interarrival time will be anomalous

• Vanguard detection scheme uses these 3 anomalies to detect pulse denial-of-service attack
Key Points

• Availability in security context deals with malicious denial of service
• Models of denial of service have waiting time policy and user agreement as key components
• Network denial-of-service attacks, and countermeasures, instantiate these models
• Amplification attacks usually hide origin of attacks, and enable flooding by an attacker that sends a relatively small number of packets