Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

Training

ANSAwise - Distributed Algorithms

Mark Madsen

Abstract

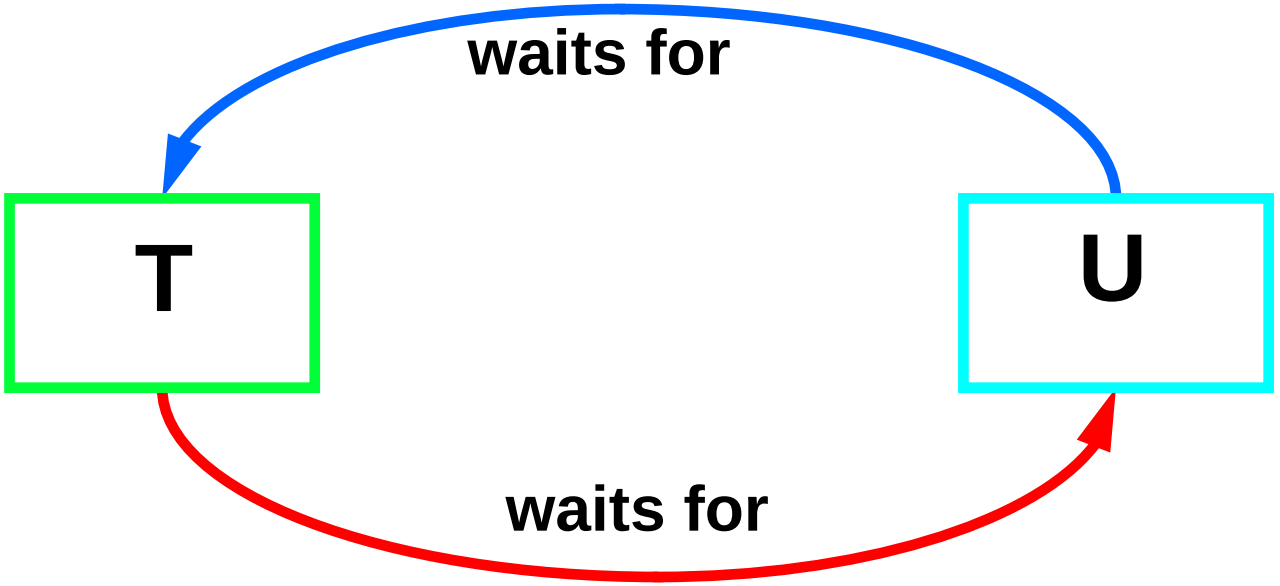
The design of distributed systems requires algorithms which are not only effective for their computational purpose, but also efficient in the context of a distributed system where their execution may be a collaborative effort between many processes and nodes. Crucial to this kind of efficiency is the robustness and scaling properties of a particular algorithm.

This module treats a representative sample of the kind of algorithms regularly used in designing for the functionality of distributed systems.

APM.1689.01 **Approved** 14th March 1996
Briefing Note

Distribution:
Supersedes:
Superseded by:

Distributed Algorithms



In this session

- **Explain the characteristics of distributed algorithms**
- **Show how some typical algorithms work**
- **Demonstrate their application to system design and analysis**

What is a Distributed Algorithm?

- A distributed algorithm is one in which different algorithm components are implemented at distinct locations
- Distributed algorithms are necessary in distributed systems
  - in infrastructure, basic services, and application services
  - to implement distributed process and transaction rules
  - to ensure correct behaviour across distributed systems

Constraints on Distributed Algorithms

- **The following are consequences of distribution:**
  - **requirement to have good scaling properties**
  - **no global consistent knowledge**
  - **no global clock time**
  - **problems of establishing consistency**
  - **need to cope with failures/reconfigurations**
  - **need to avoid distributed deadlocks**

Some Distributed Algorithms

- **Atomic broadcast**
  - **built on simpler primitives**
  - **used for reliable inter-process communication**
- **Asynchronous Update: Grapevine**
  - **used for “lazy” updating of name services**
- **Authentication: Kerberos**
  - **used for controlling access to resources**
- **Deadlock Detection: edge chasing**
  - **used for minimising resource wastage in distributed transactions**

Atomic Broadcast

- Atomic broadcast is used as a basis for fault-tolerant communications in distributed systems
- It does not rely on assumptions about network topology
- It is independent of failure modes of individual transmissions
- Atomic broadcast is constructed using simpler kinds of broadcast as building blocks
  - starting with reliable broadcast

Reliable Broadcast

- **Reliable broadcast is a broadcast with the following properties:**
  - **validity: if a correct process broadcasts a message *m*, then all correct processes eventually deliver *m*.**
  - **agreement: if a correct process delivers a message *m*, then all correct processes eventually deliver *m*.**
  - **integrity: for any message *m*, every correct process delivers *m* at most once, and only if some process has broadcast *m*.**

Reliable Broadcast Primitives

- Reliable broadcast is built on primitives `send(m)` and `receive(m)` which implement the validity and integrity properties as

- Validity: if *p* sends *m* to *q*, and both *p* and *q* and the link between them are correct, then *q* eventually receives *m*.

- Uniform integrity: for any message *m*, *q* receives *m* at most once from *p*, and only if *p* sent *m* to *q*.

Reliable Broadcast Algorithm

Every independent process p executes the following:

```
To execute broadcast(R,m):
   tag m with sender(m) and seq#(m) /* m is uniquely tagged */
   send(m) to all neighbours including p
To execute deliver(R,m):
   upon receive(m) do
      if p has not already executed deliver(R,m) then
         if sender(m) ≠ p then
            send(m) to all neighbours
         deliver(R,m)
```


Timed Reliable Broadcast Construction

- **Timed reliable broadcast is built on networks satisfying:**
  - **At most *f* processes can fail**
  - **Any two correct processes are connected by a path of at most *d* links, consisting of correct processes and links**
  - **There is a known upper bound δ on message delay**
  - **The time to execute a local step is zero**
- **Provided there are no timing failures, there is a constant Δ such that if *m* is broadcast at *t*, no correct process delivers *m* after *t*+Δ.**
- **With these assumptions, the reliable broadcast algorithm given is a Timed Reliable Broadcast with $\Delta = (f+d)\delta$.**

Atomic Broadcast Algorithm

- Let `broadcast(R,Δ,m)` be a timed reliable broadcast with delay Δ and add timestamping of messages to the algorithm already given.
- Then the following algorithm is a Timed Atomic Broadcast:

Every process p executes:

```
To execute broadcast(A,Δ,m):
    broadcast(R,Δ,m)
To execute deliver(A,Δ,m):
    upon deliver(R,Δ,m) do
        schedule deliver(A,Δ,m) at time = timestamp(m)+Δ
```


Asynchronous Update

- **Information is often replicated for fault tolerance**
- **Forcing consistency of updates does not scale well**
  - **every modification at each of N nodes generates N-1 updates**
  - **this scales quadratically, and performs poorly**
- **Asynchronous update mechanisms are used when inconsistency can be tolerated better than poor performance**

Grapevine: A Replicated Name Service

- ***State* of service**
  - **corresponds to a set of updates to distributed stored data**
- ***Observer***
  - **sees result of subset of all updates**
- ***Consistency(state, observer, value)***
  - **means that there is a subset *S* of all the updates in *state* so that executing the updates in *S* in some order gives a state in which the *observer* returns that *value***
- ***FinalState(state, update-set)***
  - **predicate to test whether there is an order in which all updates in *update-set* can be performed in order to give that *state***

Remarks on Grapevine

- The above is an example of a declarative algorithm definition
- It is possible in principle for observers to ignore all updates
- An improvement is therefore to perform consistency sweeps periodically (example: Digital's GNS - Global Name Service)
  - this is a mixed synchronous/asynchronous update strategy
  - guarantees that all clients have seen all updates up to the time of the last sweep
  - if sweeping is based on wall-clock time, there is still no effective bound on state inconsistency between clients (their inconsistency can grow arbitrarily large between sweeps)

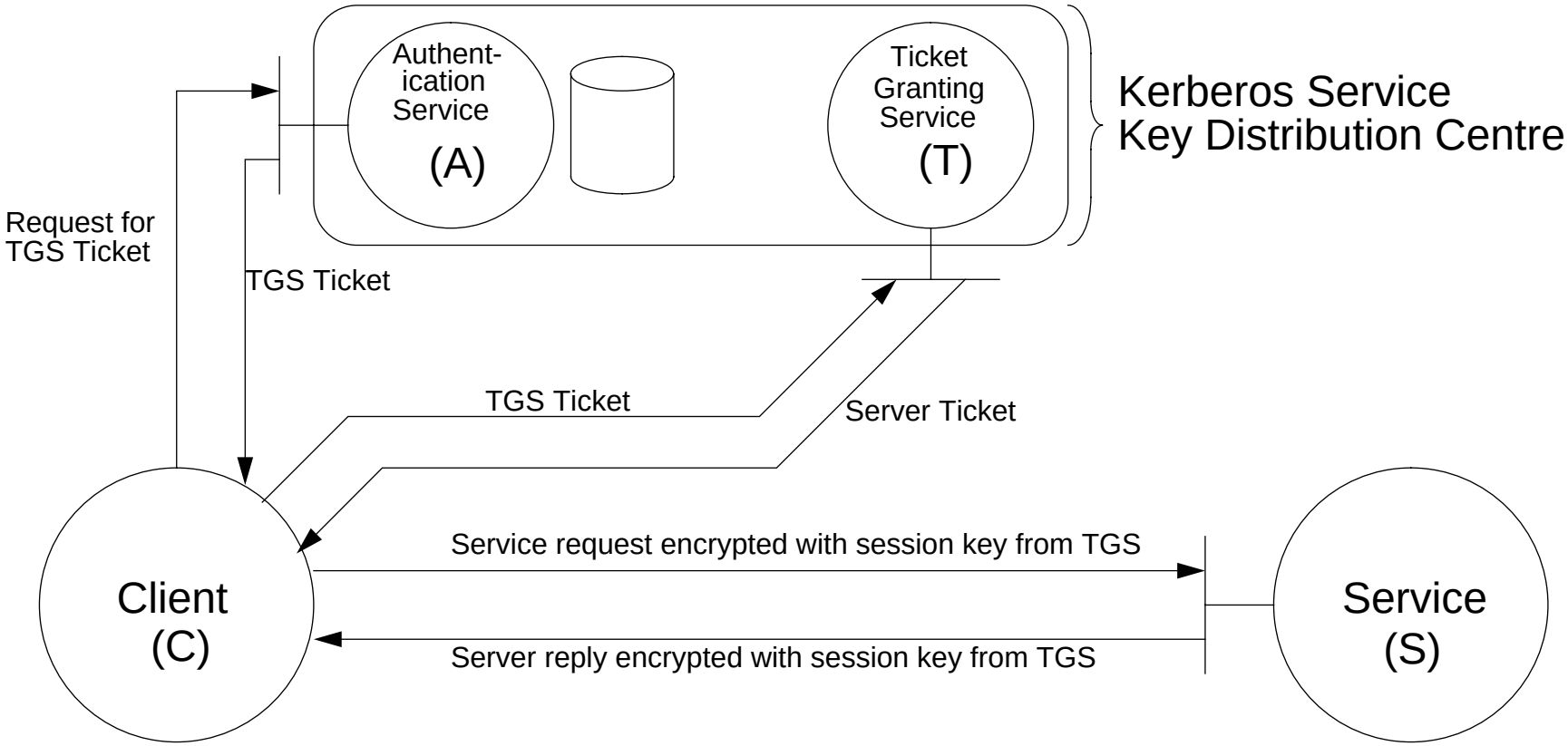
Authentication in Distributed Systems

- Authentication protocols are used to determine and control access rights to systems
- Authentication is not essentially a problem for distributed systems
  - but occurs most frequently in distributed systems because of remote access possibility
  - systems connected to open networks must assume hostile intentions on the part of users attempting to access them
- Kerberos is a widely used example of an authentication protocol

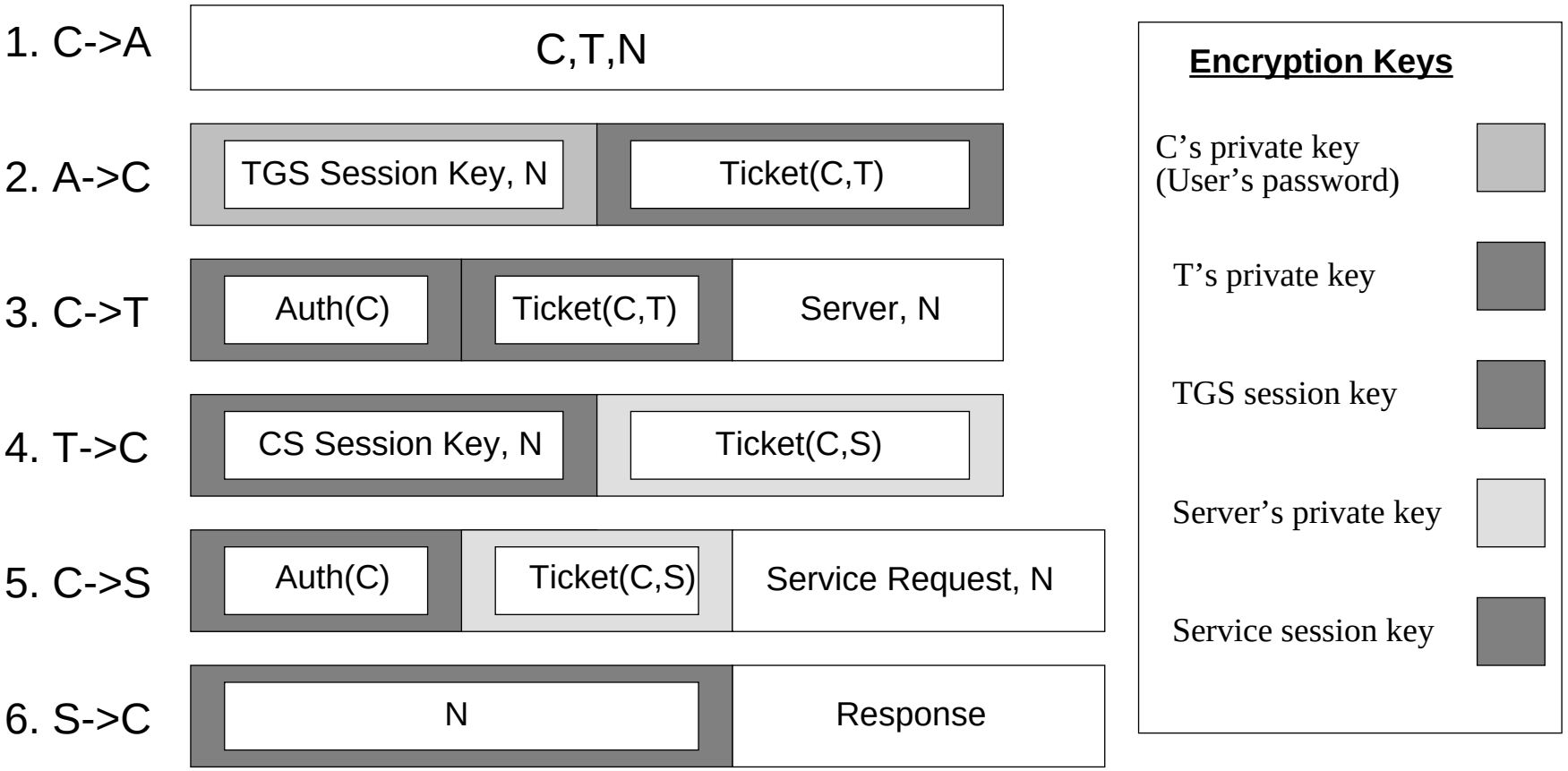
Kerberos Authentication Components

- **Authenticator**
  - sent by client to Kerberos authorisation service
  - used once
- **Ticket**
  - sent by Kerberos ticket service to client
  - tickets grant access to services
  - expires after a set period
- **Session Key**
  - sent by Kerberos to client as part of a ticket
  - used for encrypting communications with the ticketed service
  - expires when its ticket does

Authentication with Kerberos



Kerberos Protocol



Pros and Cons of Kerberos

- Kerberos uses integer timestamps as nonces
  - allows rights revocation by administration
  - guards against replay attacks
- Weaknesses of Kerberos:
  - validity of nonces depends on approximate synchronisation of clocks
  - choice of ticket expiry times trades off performance/security
  - can be spoofed by generating a race condition

Distributed Deadlock

- **Distributed deadlock is an issue whenever distributed systems use locking for concurrency control**

- **There are two possibilities for deadlock resolution**

  - **resolve deadlocks by timeouts on transactions: this is inefficient**

  - **detect deadlocks by using a scalable distributed algorithm**

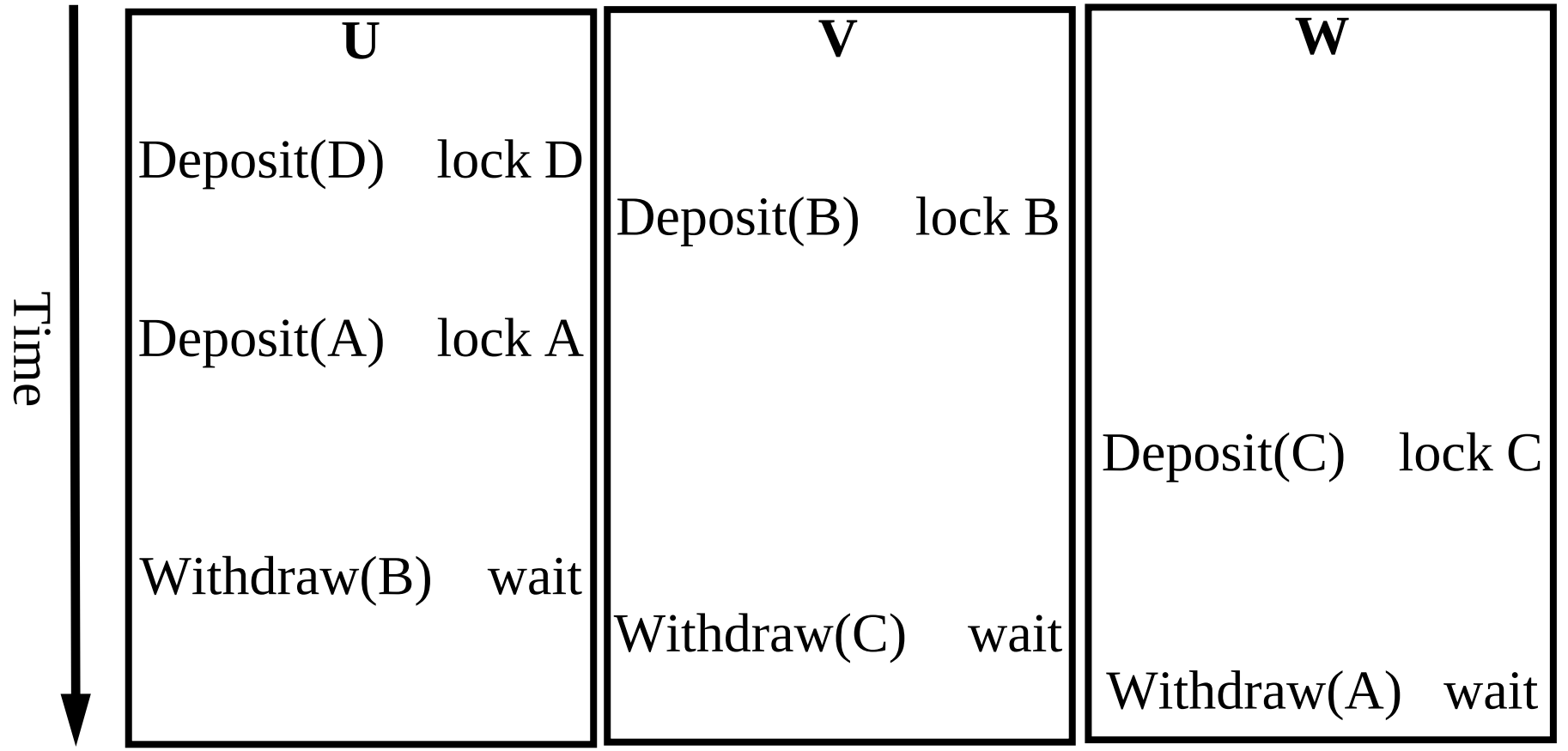
Deadlock Detection

- Deadlocks are detected by looking for cycles in the wait-for graph
- Distributed deadlock can occur when there is a cycle in the global wait-for graph
  - even when there are no cycles in the local wait-for graphs

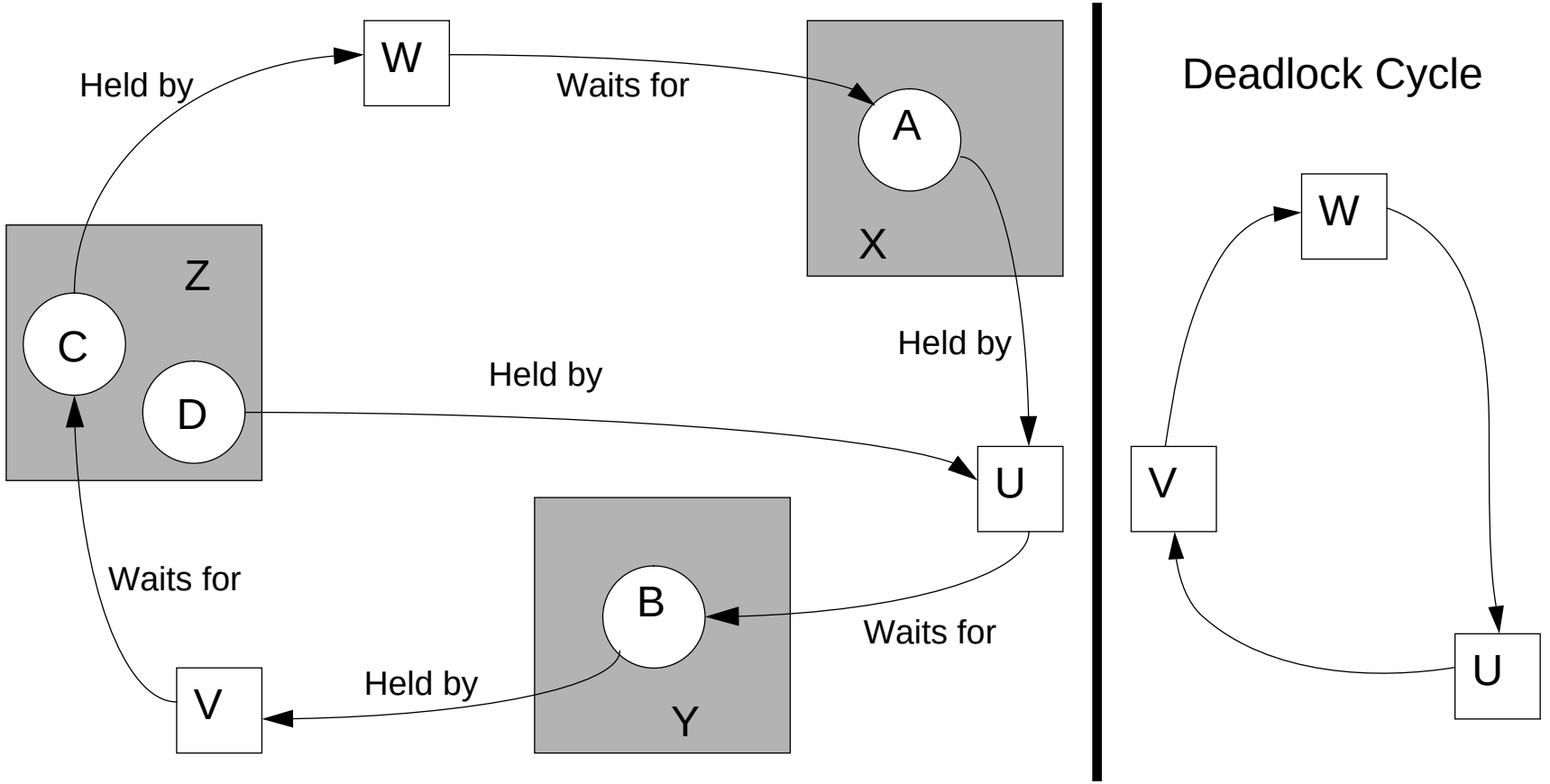
Wait-For Graphs

- **Directed graphs**
  - **nodes are either transactions or data items used by transactions**
  - **edges are data held by transactions or transactions waiting for data**

Example: Distributed Deadlock Transactions



Example: Wait-For Graph for Transactions



Deadlock Characteristics

- Deadlocks can only be detected in advance by communication between processes
- Phantom deadlocks
  - deadlocks that are resolved by the time they are detected
  - possible because of delay in gathering wait-for information in one place
  - can occur if transactions abort during detection phase
  - Two-Phase Commit is free of phantom deadlocks - often used for this reason

Edge-Chasing Algorithms

- **Edge-chasing defines a class of algorithms using 3 phases**
  - **initiation**
  - **detection**
  - **resolution**

Initiation Phase

- Server notes transaction T is waiting for U, where U is waiting for data D at another node.
- Server initiates detection by sending a *probe* to the server controlling the data D
  - the probe contains the edge <T-V>
- If U is sharing a lock, probes are sent to all holders of the lock

Detection Phase

- Server controlling data D receives probe <T-V>
- It checks if U is waiting on another transaction
- If U is waiting on transaction V where V is waiting on data P:
  - the server forwards new probe <T-U-V> to node controlling P

Resolution Phase

- When a server receives a probe containing a cycle, it either
  - aborts its own transaction from the cycle contained in the probe, or
  - initiates negotiation with the other servers to choose a transaction to be aborted

Remarks on Edge Chasing

- **Edge chasing is a true distributed algorithm**
  - **unlike the alternative of assembling all the wait-for information at a designated central server**
- **The distributed property of edge-chasing is that any of the servers can be the one to detect the deadlock cycle**
  - **no central deadlock-detecting service is required**

Summary

- **Distributed algorithms are fundamentally different from centralised ones**
- **Important issues are completion, fault tolerance, and scaling**
- **Distributed algorithms often involve tradeoffs, such as**
  - **robustness against speed (like edge chasing)**
  - **consistency against speed (like asynchronous update)**

Finding Out More

- For more on distributed transactions and consistency, see
  - *Transaction Processing: Concepts and Techniques,* by Jim Gray & Andreas Reuter (Morgan Kaufmann 1994)
- For algorithms for replication, concurrency, and fault tolerance, see
  - *Distributed Systems: Concepts and Design*, by George Coulouris, Jean Dollimore, & Tim Kinderberg (Addison-Wesley 1994)
- For more on a variety of distributed algorithms, see the contributions published in
  - *Distributed Systems, 2ed*, edited by Sape Mullender (Addison-Wesley 1993)