
Introduction to Distributed Algorithms

21 February 2011
Lecture 1

Slide credits: R. H. Mak and P. Veltkamp

Topics for Today

- Distributed systems and algorithms
 - What, why, where
- (A)Synchronicity
 - For cooperation
 - For communication
- Programming language
 - Control-oriented
 - Event-driven
- Graph theory (quick overview)
 - Basic definitions
 - Special graphs

Source: Tel 1

Distributed systems (what)

A distributed system consists of:

- Processing elements
 - Autonomous
 - Full-fledged computer
 - Heterogeneous
- Communication subsystem or network
 - Point-to-point channels
 - Message passing
 - Usually asynchronous, but sometimes synchronous
 - Modeled as a graph
 - Subject to dynamic changes

Distributed systems (why)

1. Unavoidable
 - inherent distributed asynchronous environment
1. Information exchange
2. Resource sharing
 - Services, peripherals, ...
1. Reliability improvement
 - Replication of data
 - Removal single point of failure
1. Performance enhancement
 - Due to concurrency, load balancing
 - Removal of bottlenecks through replication
1. Design simplification
 - Modularity
 - Specialization

Distributed algorithm (what)

Distributed algorithms are algorithms that address the fundamental problems of distributed systems:

1. limited local knowledge
 1. number of processor
 2. network topology
2. Asynchrony
 1. processor speed
 2. communication speed
3. Failure
 1. of data links
 2. of processors

Distributed algorithm characteristics

1. Lack of knowledge (existence) of a global state
 - Due to local connectivity, and asynchronous mode of communication
1. Lack of a global time frame
 - Also called asynchronous mode of cooperation
1. Non-determinism
 - Differences in execution speed
 - Varying communication delays, caused by:
 1. Congestion
 2. multi-path routing
 3. atmospheric conditions (for wireless links)
 4. mobility of processors
 - Extreme cases of the above are link or processor failures

Distributed algorithms (where)

1. Network protocols
 1. Reliable connections
 2. Routing
2. Transaction systems
 1. Banking systems
 2. Reservation systems
 3. transactions must be atomic
3. Distributed databases
 1. Concurrency-control
 2. Consistency of replicated data
4. Distributed operating systems
 1. Termination/deadlock detection, recovery
 2. Synchronization, resource sharing
 3. Distributed shared memory
5. Middleware

Algorithmic problem classes

1. Broadcasting
2. Routing
3. Synchronization
4. Resource (al)location
5. Mutual exclusion
6. Termination detection
7. Deadlock detection
8. Leader election
9. Consensus

Synchrony

1. Mode of communication

1. Asynchronous:

1. non-blocking send, blocking receive
2. deadlock, when both parties wait for input

2. Synchronous:

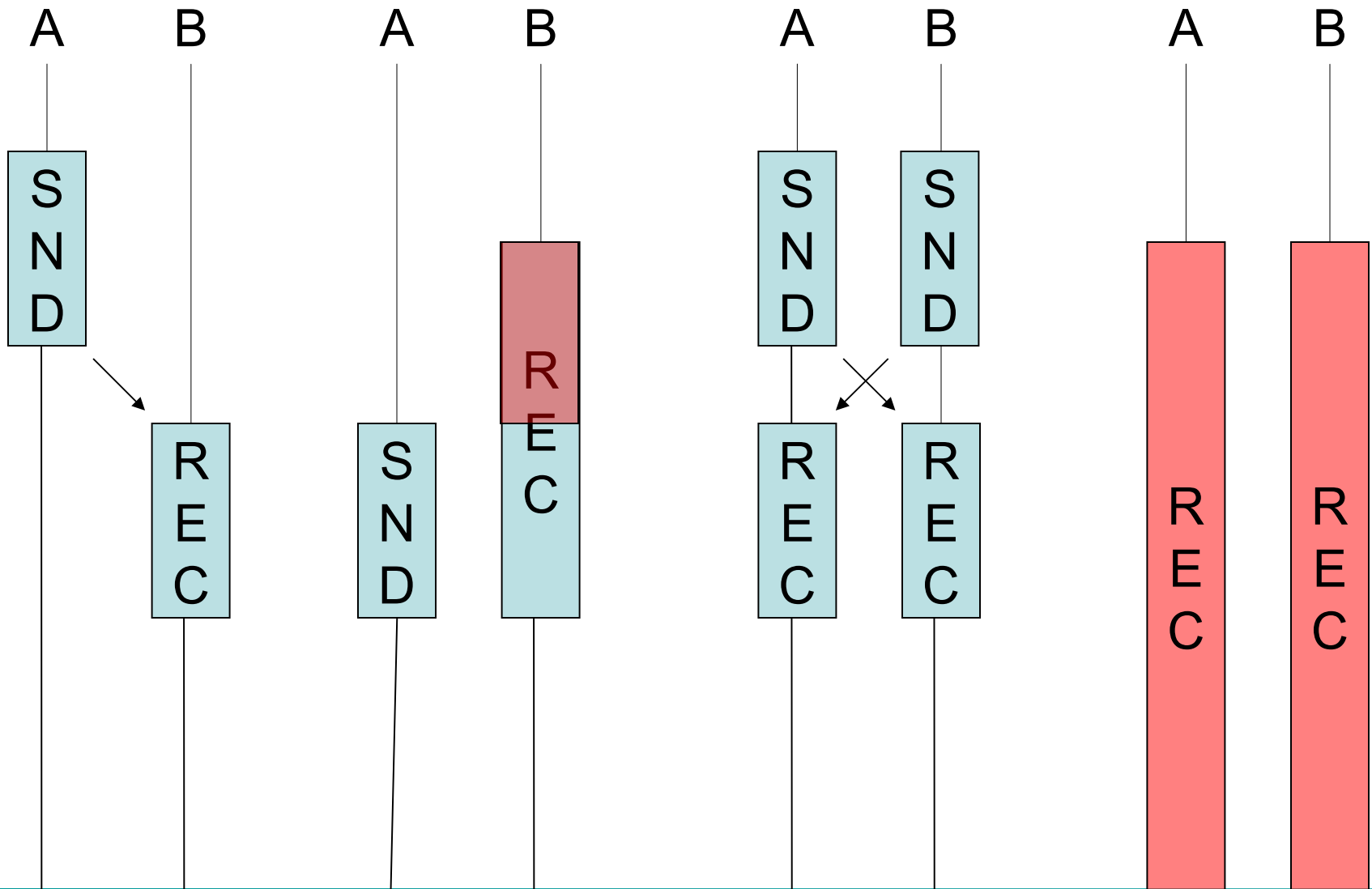
1. first participant (sender or receiver) blocks until its counterpart is ready; simultaneous completion
2. deadlock, when parties disagree on order of communications

2. Mode of cooperation

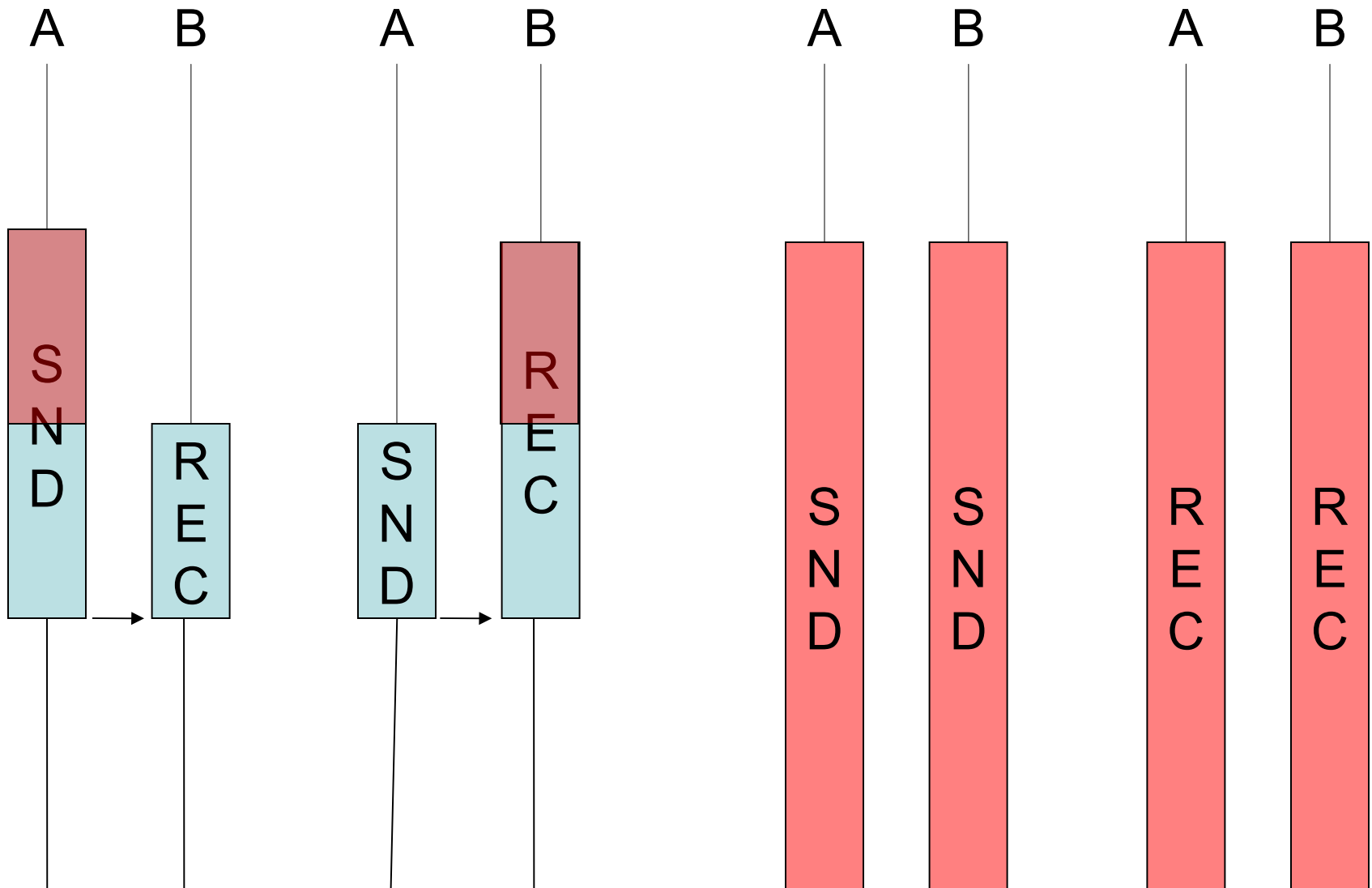
1. Synchronous network model

1. there is a notion of global time
2. processes operate in lockstep
3. reduction of number of communications

Asynchronous communication



Synchronous communication



Synchronous cooperation

- Unrealistic in distributed systems
 - Can be simulated by asynchronous networks
 - First step towards development of an asynchronous solution
- Powerful model
 - Reduces number of communications
 - Notion of global time can be used to communicate any integer number using two bits.
 - Cellular automata
 - Used for consensus algorithms
 - Muddy children puzzle, Firing squad problem

Pseudo-code conventions 1/2

1. Data

1. Set variables are allowed; in particular in process p variable $Neigh_p$ is used to indicated the set of neighbors of p .
2. Set operators are used in boolean and arithmetic expressions.

2. Messages

1. Tuple structure: type field obligatory, zero or more data fields
<type, data, data>
1. Pattern matching (on the type field): $msg :: <T, a, b>$

1. Flow of control

- Pascal-like + unordered execution:
forall condition do statement

Pseudo-code conventions 2/2

1. Messages

1. Sending of messages: **send** *msg* to *dest*

shout *msg* \equiv forall $q \in Neigh_p$ do send *msg* to q

1. Receipt of messages:

1. Both *msg* and *src* are var-parameters

receive *msg* from *src*

1. *msg* a var-parameter and *src* a val-parameter

receive *msg* from **this** *src*

• Probe: boolean expression with pattern matching

<T, a, b> is available on *q*

Programming styles

Control-oriented

- Program consists of a variable declaration part followed by a statement list.
- Local algorithm consists of a single execution of the statement list.
- Non-determinism explicit in program text:
 1. Choice on input: leave sender unspecified.
 2. Choice between input and output: requires probe.

Event-driven

- Program consists of a variable declaration part followed by a set of actions. An action is a statement list guarded by a boolean expression (commonly arrival of a message).
- An action is enabled, when its guard evaluates to true.
 - Beware the stability of the guards
- The local algorithm consists of repeated non-deterministic selection and subsequent execution of an enabled action.
 - Selection is not guaranteed to be fair.

Example: and-gate

```
var  $sa_p$  : boolean init false;  
     $sb_p$  : boolean init false;
```

```
begin while true do
```

```
    begin if <up> is available on  $in_a$  then
```

```
        begin  $sa_p := true$ ; receive <up> from  $in_a$  end;
```

```
    if <down> is available on  $in_a$  then
```

```
        begin  $sa_p := false$ ; receive <down> from  $in_a$  end;
```

```
    if <up> is available on  $in_b$  then
```

```
        begin  $sb_p := true$ ; receive <up> from  $in_b$  end;
```

```
    if <down> is available on  $in_b$  then
```

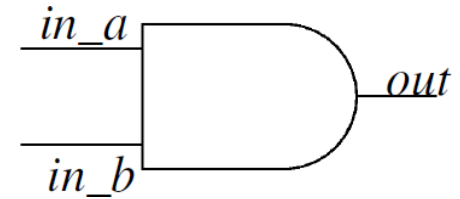
```
        begin  $sb_p := false$ ; receive <down> from  $in_b$  end;
```

```
    if  $sa_p$  AND  $sb_p$  then send <up> to out
```

```
    else send <down> to out
```

```
    end
```

```
end
```



Example: and-gate

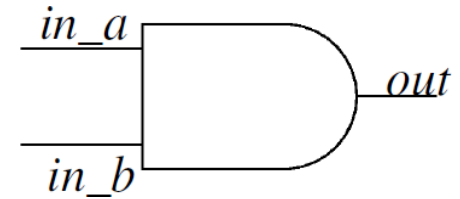
```
var  $sa_p$  : boolean init false;  
     $sb_p$  : boolean init false;
```

```
{ a message  $m$  arrives from  $in_a$  }  
begin receive  $m$  from  $in_a$ ;  
    if  $m \dots \langle up \rangle$  then  $sa_p := true$  else  $sa_p := false$   
end;
```

```
{ a message  $m$  arrives from  $in_b$  }  
begin receive  $m$  from  $in_b$ ;  
    if  $m \dots \langle up \rangle$  then  $sa_b := true$  else  $sa_b := false$   
end;
```

```
{ $sa_p$  AND  $sa_b$ }  
begin send  $\langle up \rangle$  to out end
```

```
{! $sa_p$  OR ! $sa_b$ }  
begin send  $\langle down \rangle$  to out end
```



Question: do the
two and-gates have
the same behavior?

Elementary graph theory

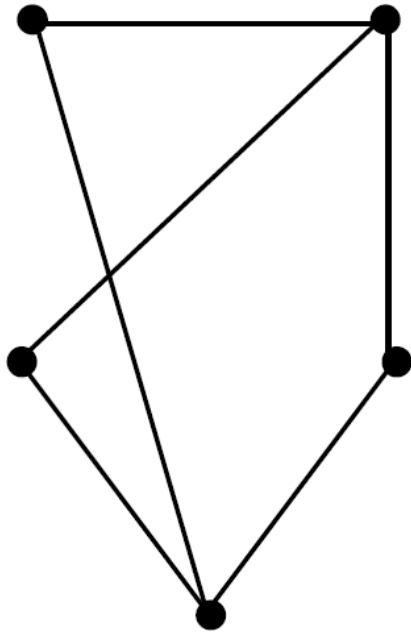
Concepts, definitions:

1. Directed, undirected, weighted graphs
2. Paths, cycles, diameter
3. Connectivity

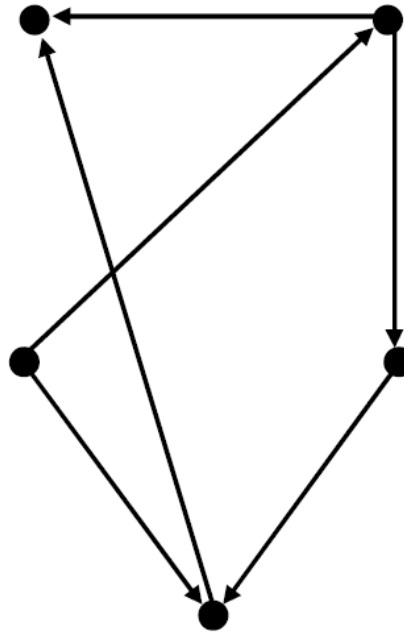
Special graphs:

1. Definitions
2. Properties

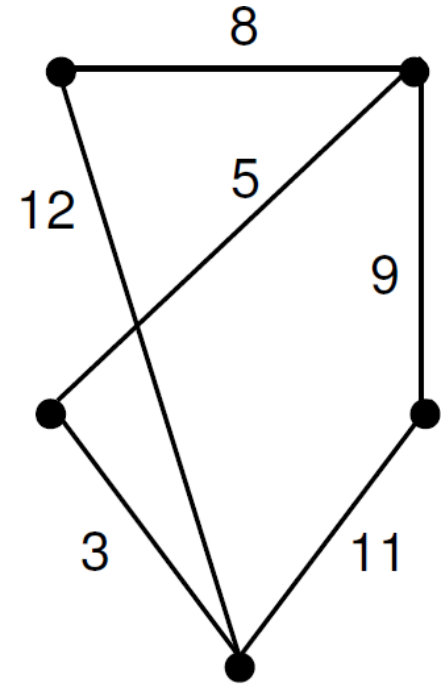
Examples of graphs



Undirected



Directed

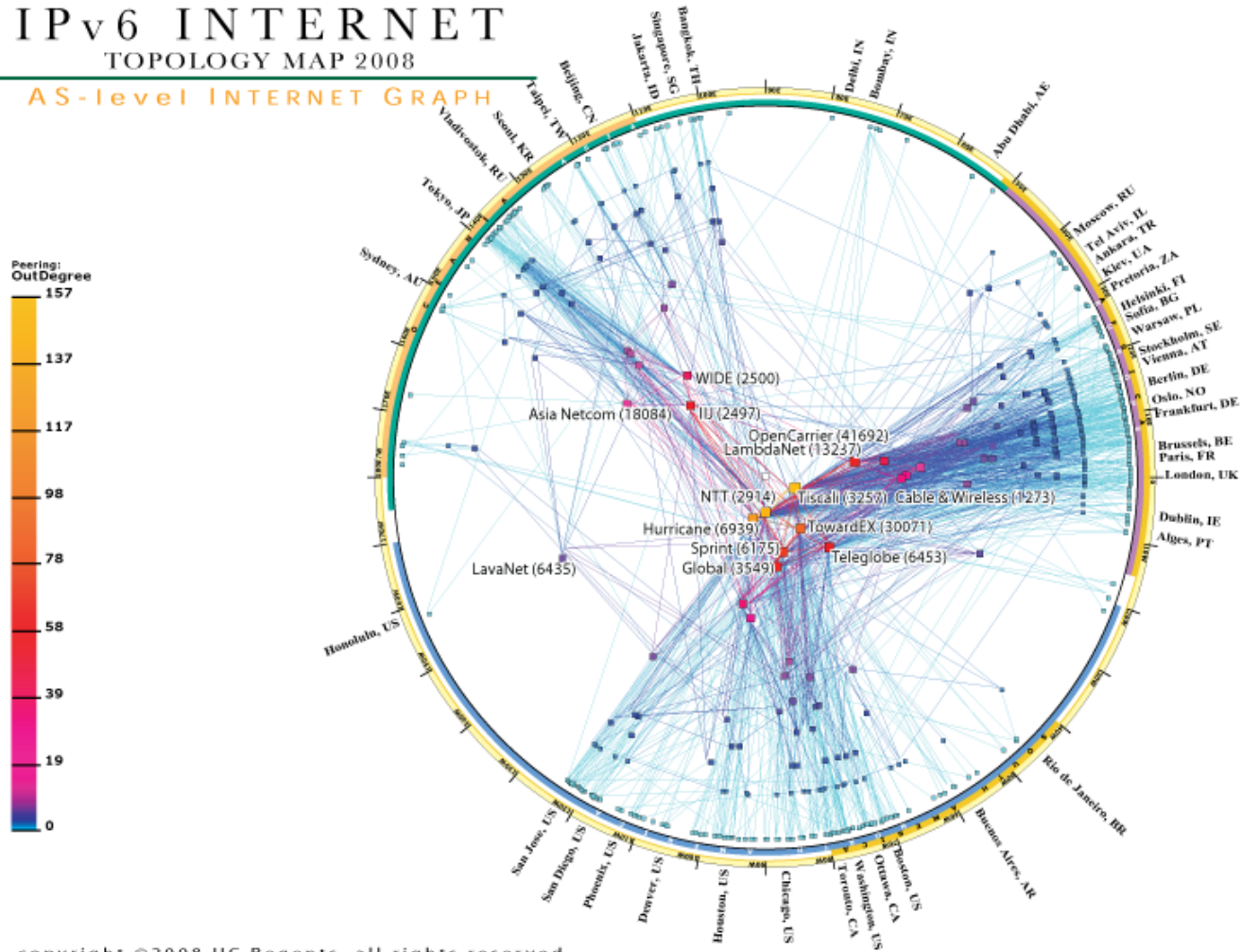


Undirected
Weighted

Internet AS-graph

IPv6 INTERNET TOPOLOGY MAP 2008

AS-level INTERNET GRAPH



Undirected graphs

An *undirected graph* G is a pair (V, E) , where

- V is the *node set* of G
- $E \subseteq \{ \{u, v\} \mid u, v \in V \}$ is the *edge set* of G .
 - edges are unordered pairs
 - instead of $\{u, v\}$ we also write uv , or equivalently, vu
 - no autoloops $\{u, u\}$
 - if $uv \in E$, then nodes u and v are *adjacent*, and uv is an *incident* edge of both u and v

For $v \in V$ we define the set of *neighbors* of v by

- $Neigh_v = \{ u \mid uv \in E \}$
- $|Neigh_v|$ is the *degree* of v

Directed graphs

A *directed graph* G is a pair (V, E) , where

- V is the *node set* of G .
- $E \subseteq V \times V$ is the *edge (arc, arrow) set* of G .
 - edges are ordered pairs
 - instead of (u, v) we also write uv
 - $uv \in E$ is an *outgoing edge* of u and an *incoming edge* of v
 - no autoloops (u, u)

For $v \in V$ we define the sets

- $In_v = \{ u \mid uv \in E \}$ of *in-neighbors*, or *predecessors*,
- $Out_v = \{ u \mid vu \in E \}$ of *out-neighbors*, or *successors*.
- $|In_v|$ is the *in-degree* and $|Out_v|$ is the *out-degree* of v ,
- $Deg(v) = |In_v| + |Out_v|$ is the *degree* of v .