

Pregel: A System for Large Scale Graph Processing

Malewicz, et.al

Presenter: Stephan Brandauer

This talk:
- High level intro

2010 ACM SIGMOD
International Conference on
Management of Data

Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn,
Naty Leiser, and Grzegorz Czajkowski
Google, Inc.
{malewicz,austern,ajcbik,dehnert,ilan,naty,gczaj}@google.com

ABSTRACT

Many practical computing problems concern large graphs. Standard examples include the Web graph and various social networks. The scale of these graphs—in some cases billions of vertices, trillions of edges—poses challenges to their efficient processing. In this paper we present a computational model suitable for this task. Programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology. This vertex-centric approach is flexible enough to express a broad set of algorithms. The model has been designed for efficient, scalable and fault-tolerant implementation on clusters of thousands of commodity computers, and its implied synchronicity makes reasoning about programs easier. Distribution-related details are hidden behind an abstract API. The result is a framework for processing large graphs that is expressive and easy to program.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming—*Distributed programming*; D.2.13 [Software Engineering]: Reusable Software—*Reusable libraries*

General Terms

Design, Algorithms

Keywords

Distributed computing, graph algorithms

1. INTRODUCTION

The Internet made the Web graph a popular object of analysis and research. Web 2.0 fueled interest in social networks. Other large graphs—for example induced by transportation routes, similarity of newspaper articles, paths of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies include notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and may incur additional charges. Copyright 2010 ACM 978-1-4558-5424-7/10/00...\$10.00.

disease outbreaks, or citation relationships among published scientific work—have been processed for decades. Frequently applied algorithms include shortest paths computations, different flavors of clustering, and variations on the page rank theme. There are many other graph computing problems of practical value, e.g., minimum cut and connected components.

Efficient processing of large graphs is challenging. Graph algorithms often exhibit poor locality of memory access, very little work per vertex, and a changing degree of parallelism over the course of execution [31, 39]. Distribution over machines exacerbates the locality issue, and increases the probability that a machine will fail during computation. Despite the ubiquity of large graphs and their commercial importance, we know of no scalable general-purpose systems for implementing arbitrary graph algorithms over arbitrary graph representations in a large-scale distributed environment.

Implementing an algorithm to process a large graph locally means choosing among the following options:

1. Crafting a custom distributed infrastructure requiring a substantial implementation effort to be repeated for each new algorithm or graph representation.
2. Relying on an existing distributed computing infrastructure, which is often ill-suited for graph processing. MapReduce, for example, is a very good fit for a wide range of scale computing problems. It is sometimes used to mine large graphs [11, 30], but this is not an optimal performance and usability model for processing data that has been aggregated [41] and SQL-like models for processing data have been proposed [41] but these extensions are usually not graph algorithms that often better fit a message-passing model.
3. Using a single-computer graph algorithm as BGL [43], LEDA [35], NetworkX [29], Stanford GraphBase [29], or Pregel [22], or a library of problems that can be adapted to a distributed system.
4. Using an existing parallel graph algorithm as BGL [22] and CGMgraph [8] or other issues that are important in distributed systems.

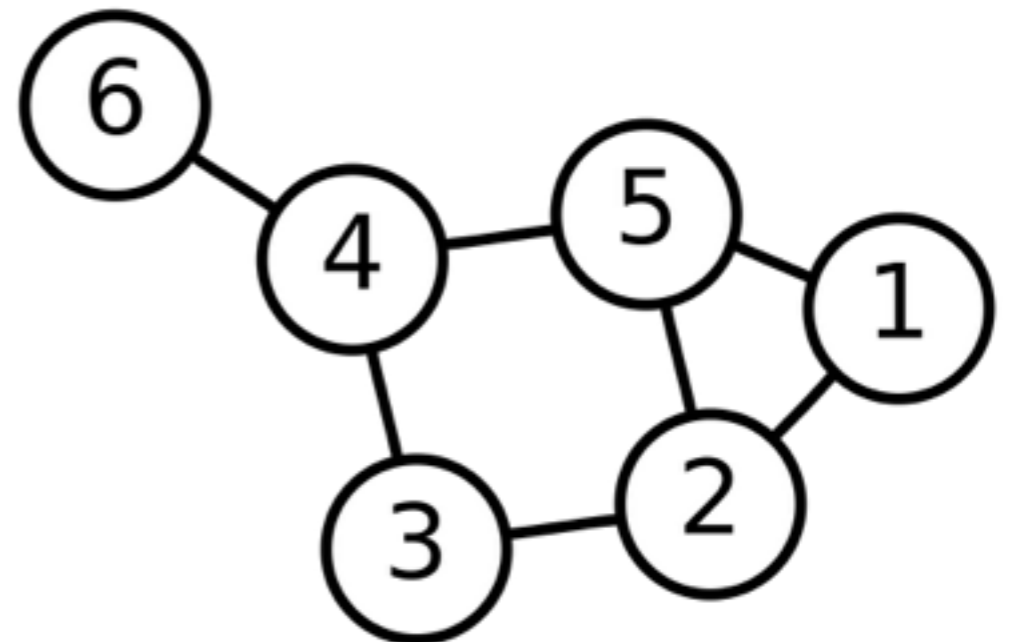
None of these alternatives fit our requirements for distributed processing of large scale graphs.

I like high level abstractions that carefully exploit their semantics for optimisations.

Pregel is a particularly pretty case.

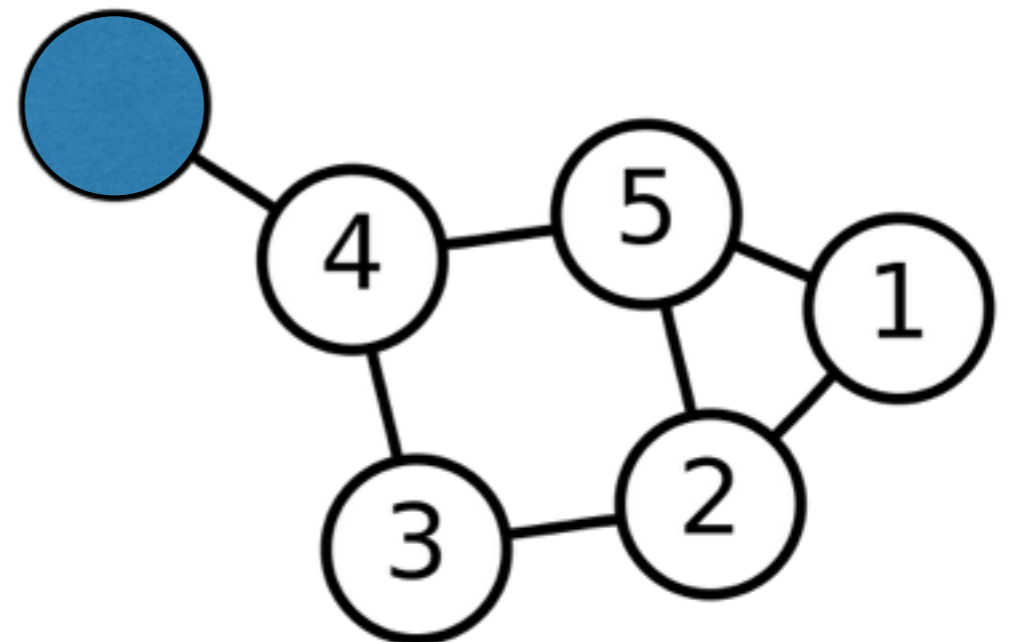
Graph Processing: hard to **parallelise**

Graphs are connected by nature — it's easy to get data races.



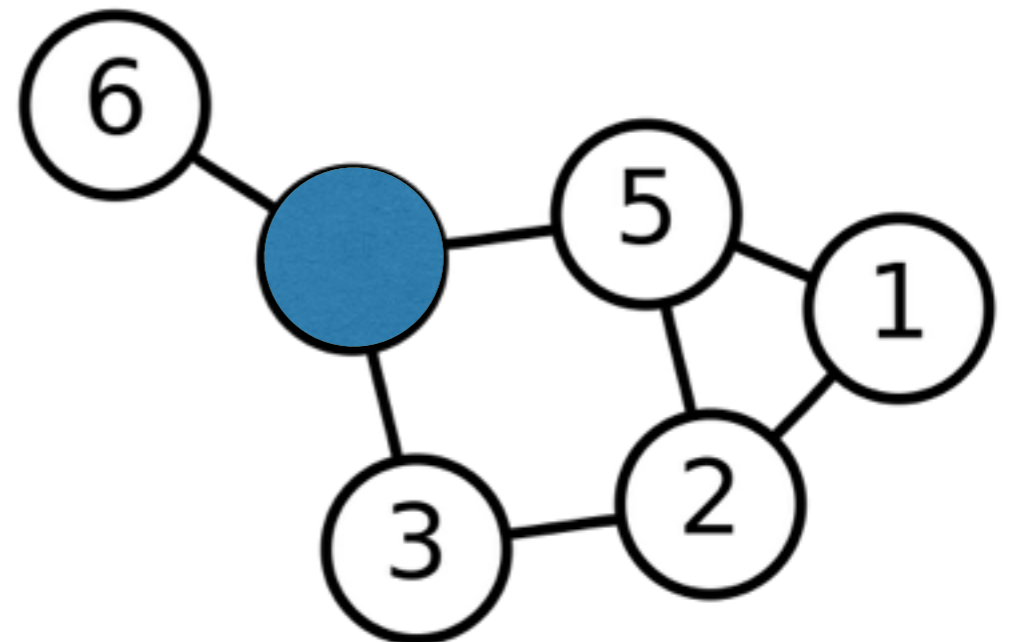
Graph Processing: hard to **parallelise**

Graphs are connected by nature — it's easy to get race conditions!



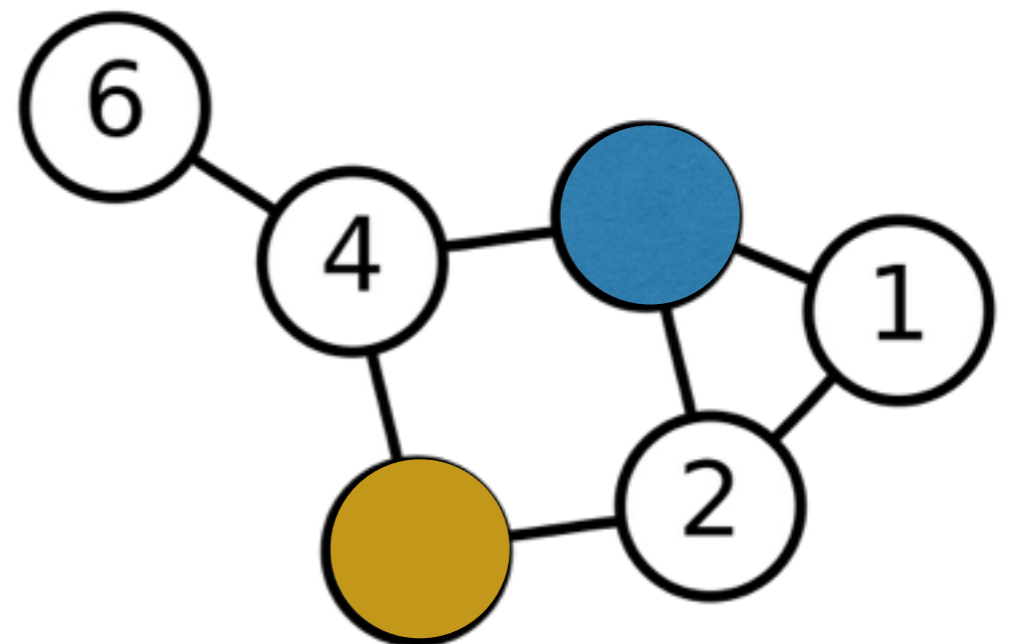
Graph Processing: hard to **parallelise**

Graphs are connected by nature — it's easy to get race conditions!



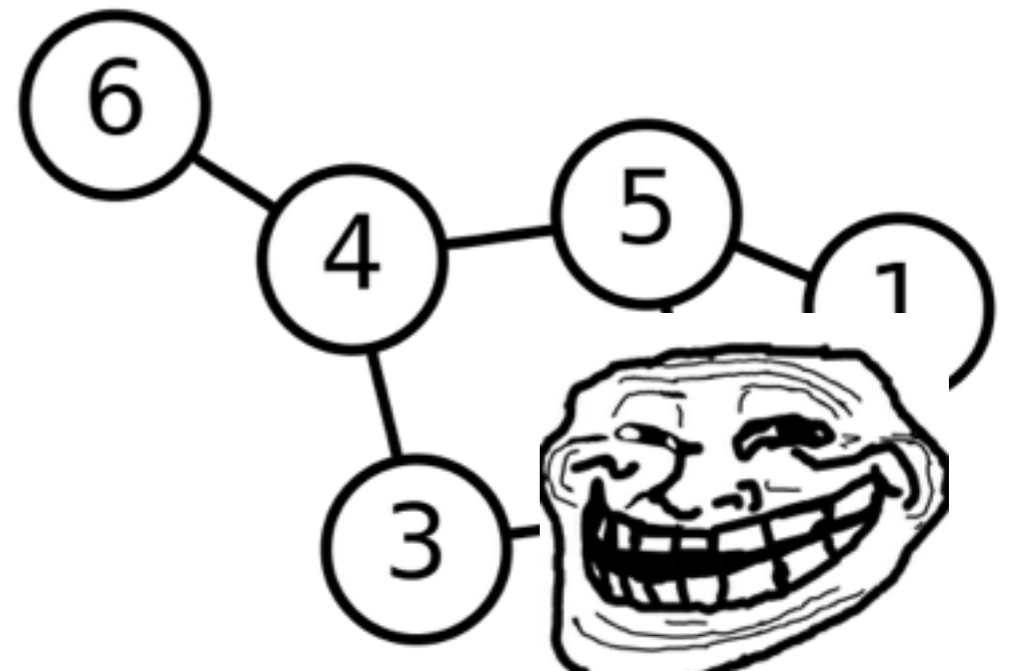
Graph Processing: hard to **parallelise**

Graphs are connected by nature — it's easy to get race conditions!



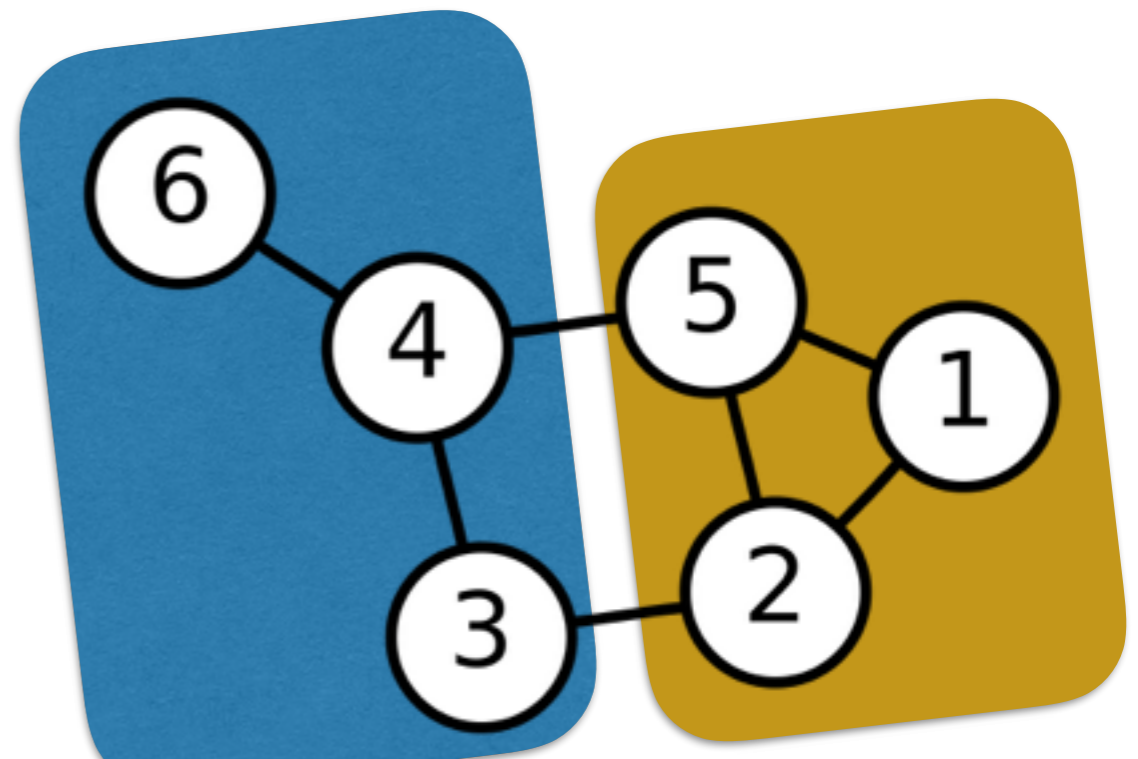
Graph Processing: hard to **parallelise**

Graphs are connected by nature — it's easy to get data races.



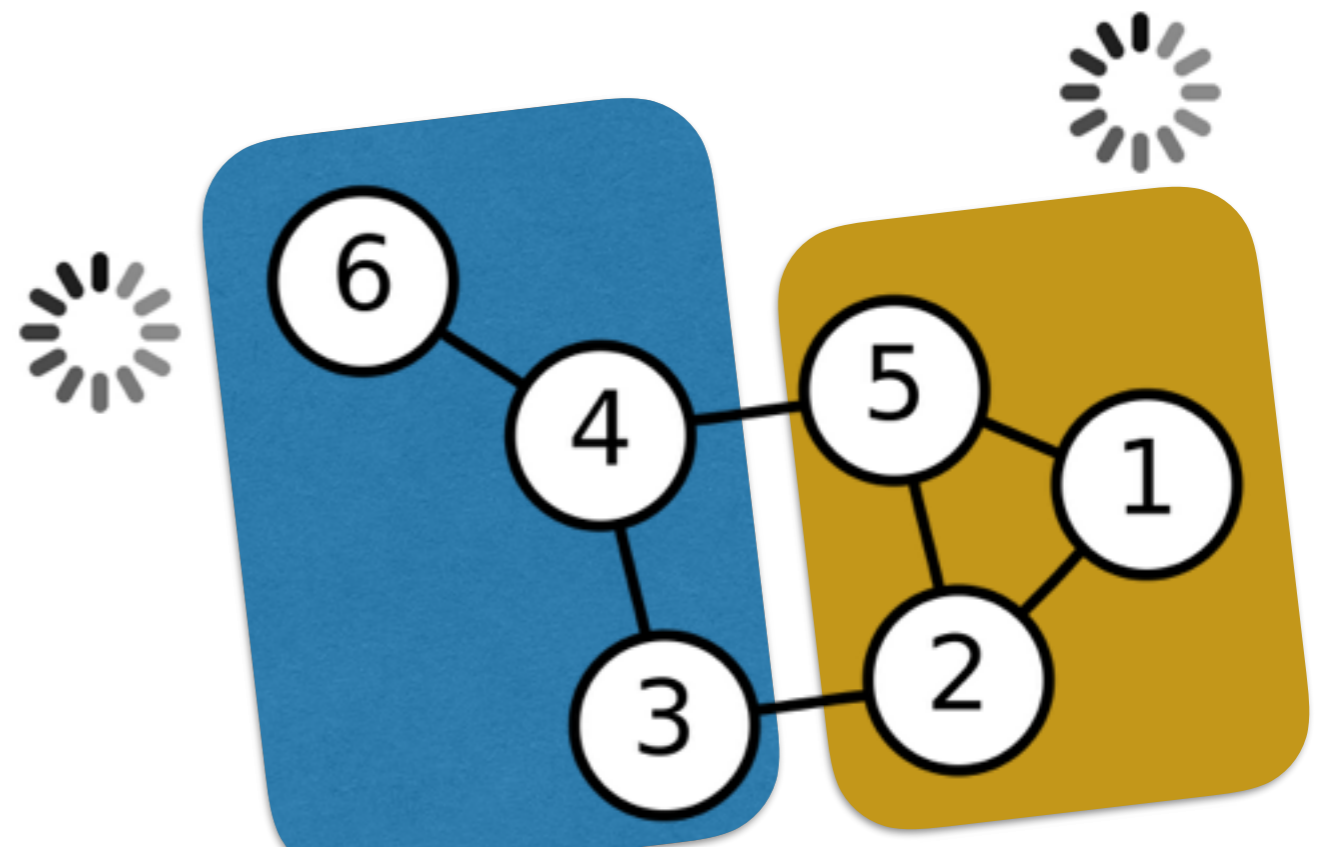
Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



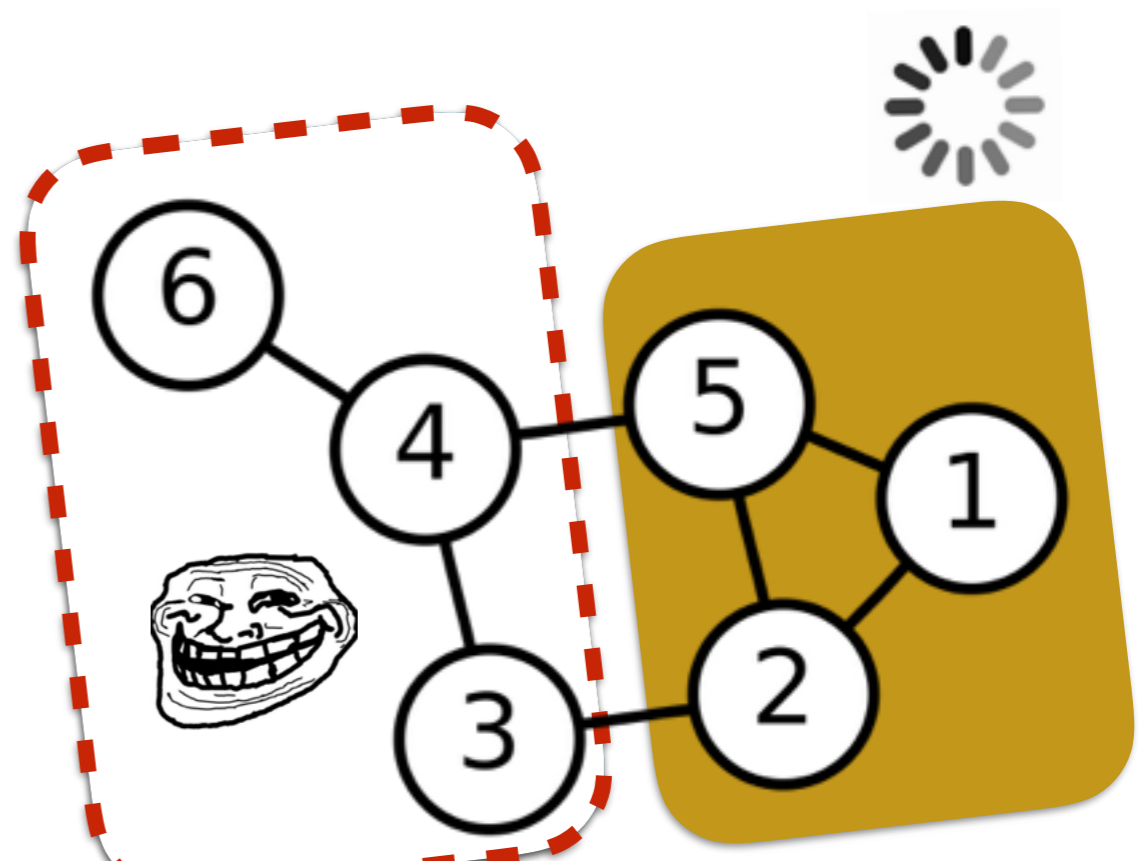
Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



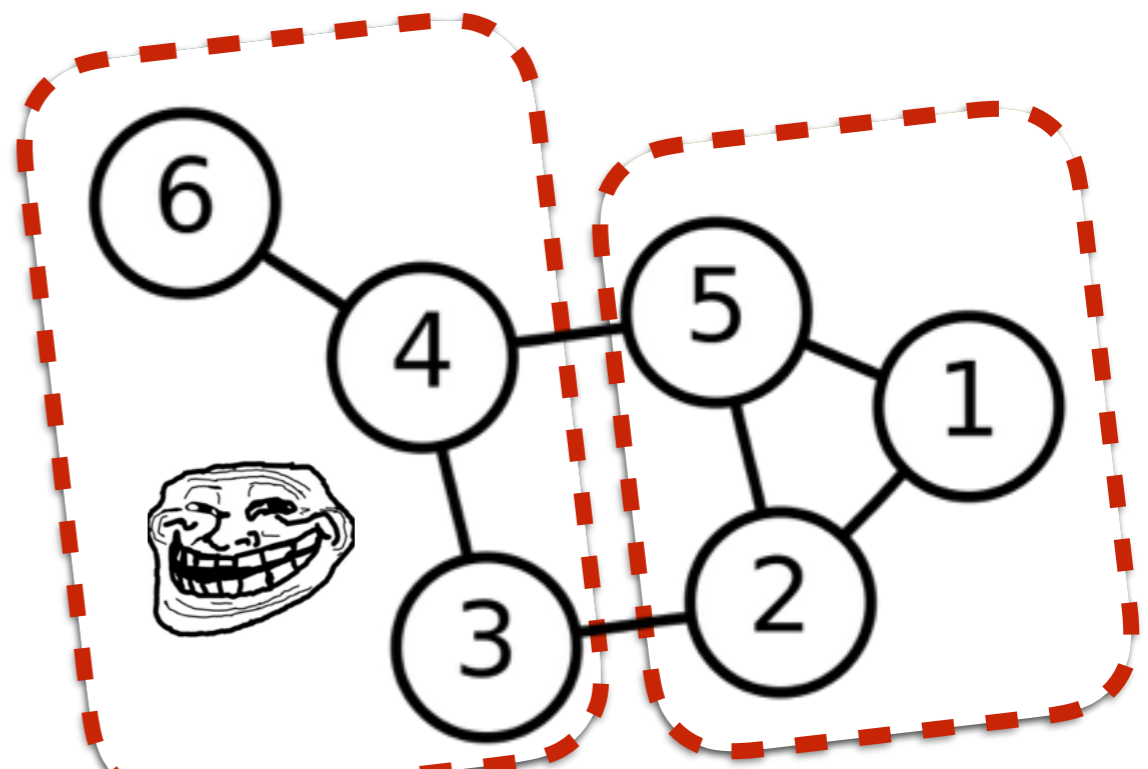
Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



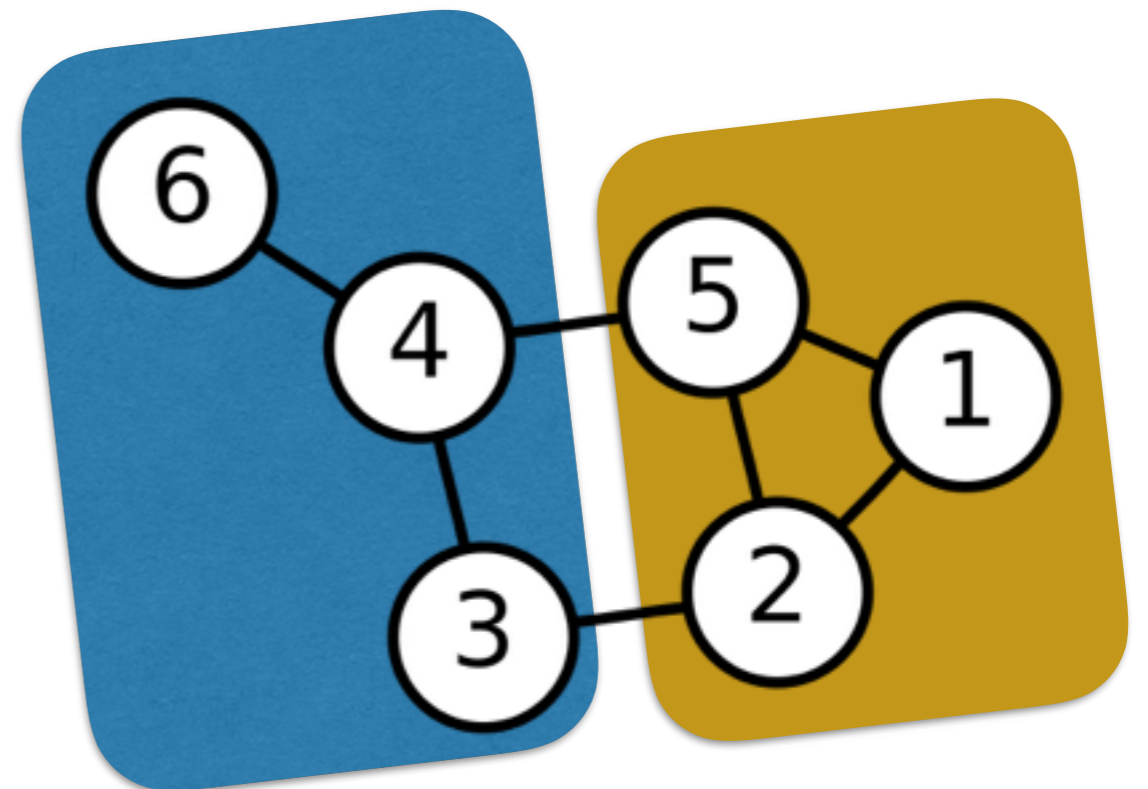
Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



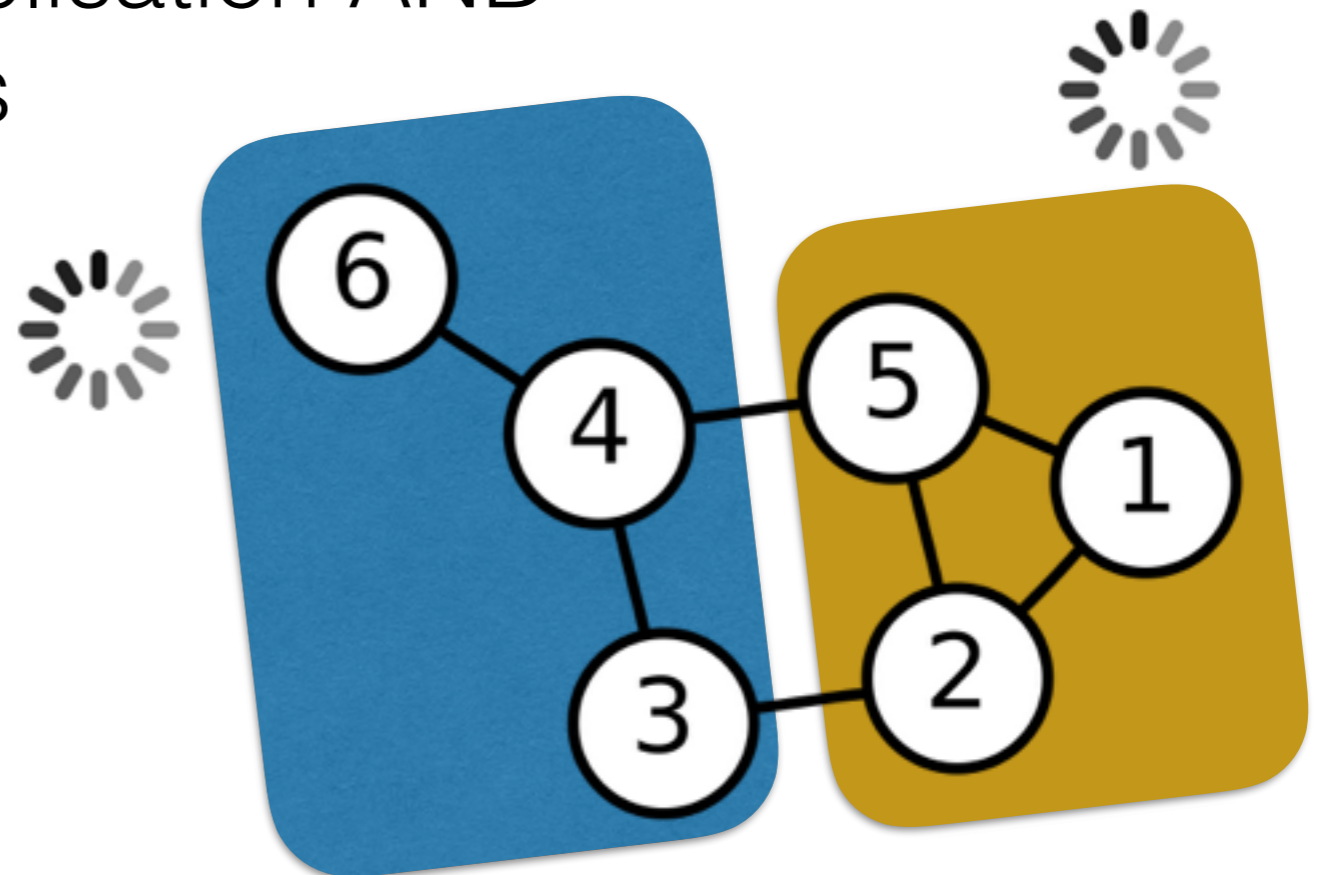
Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



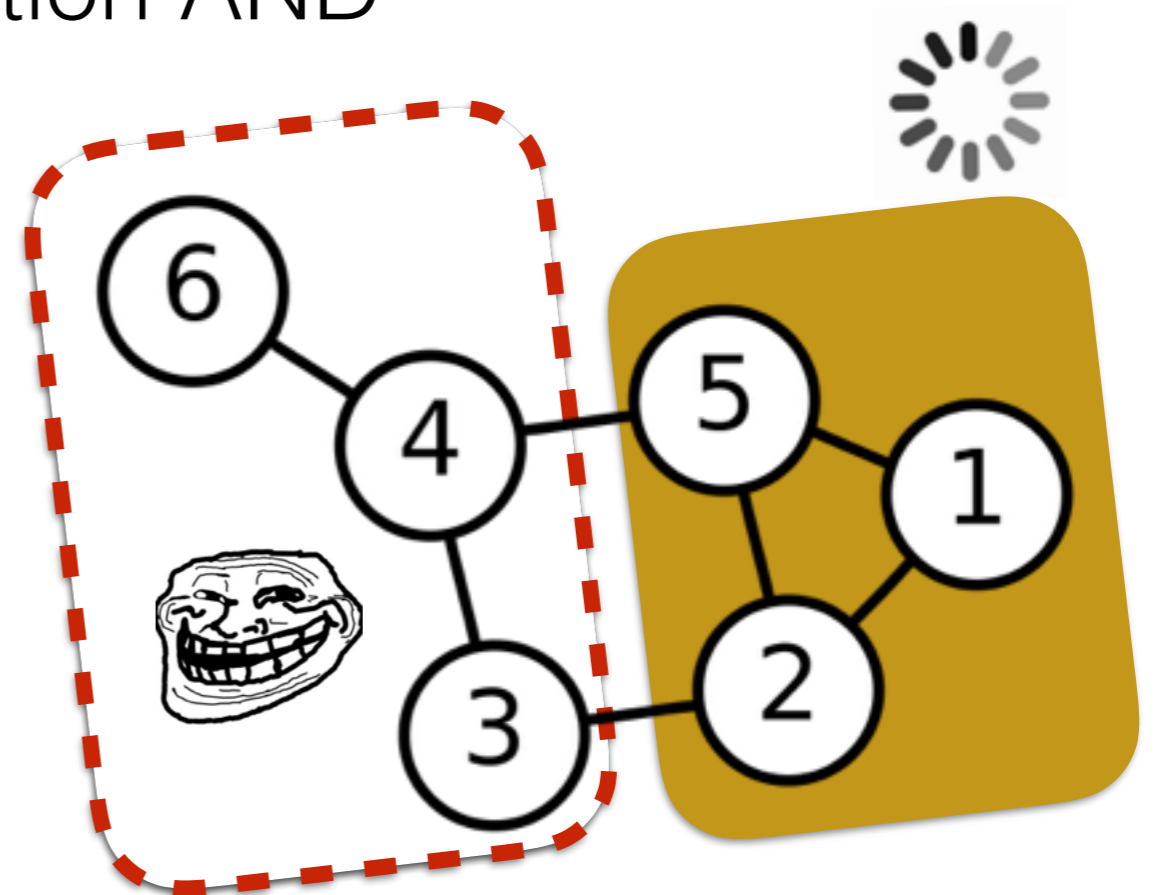
Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



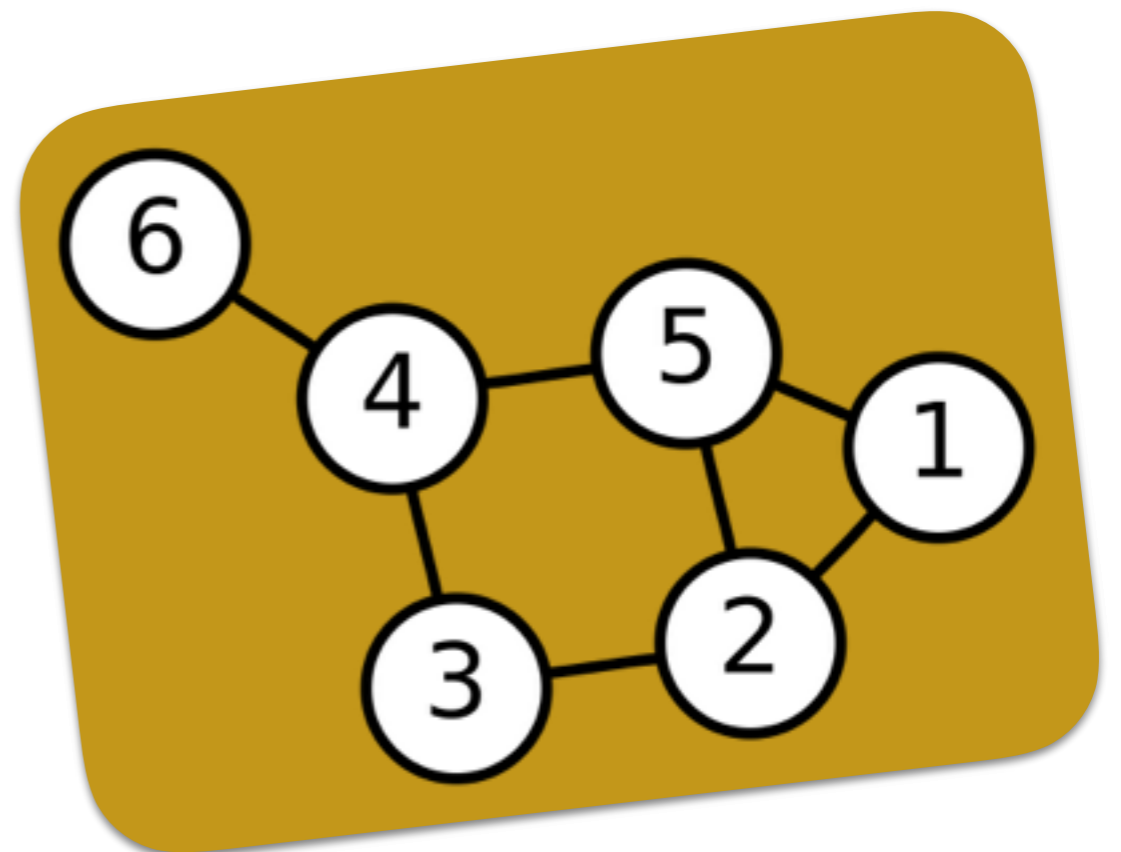
Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



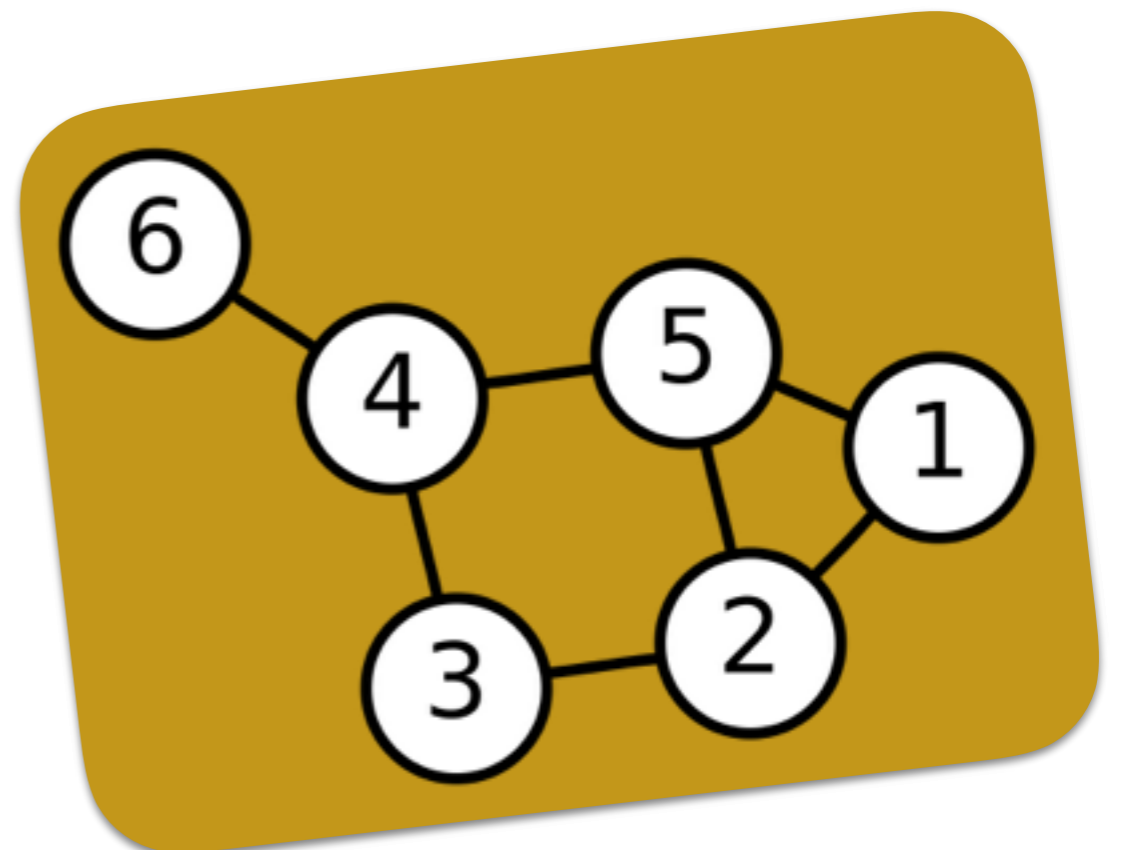
Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



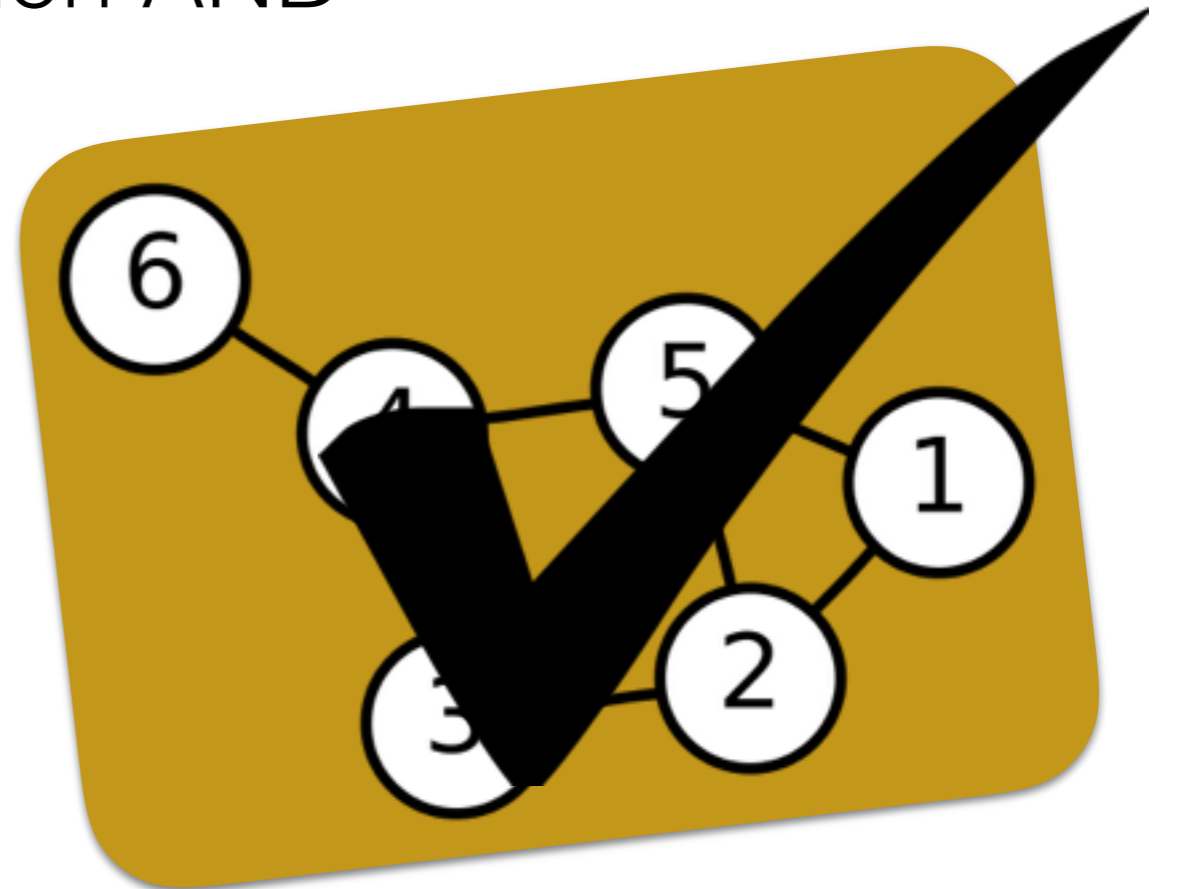
Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



Two Birds



half-caf



double espresso

Two Birds



race conditions



double espresso

Two Birds



race conditions



fault tolerance

One Stone



Pregel

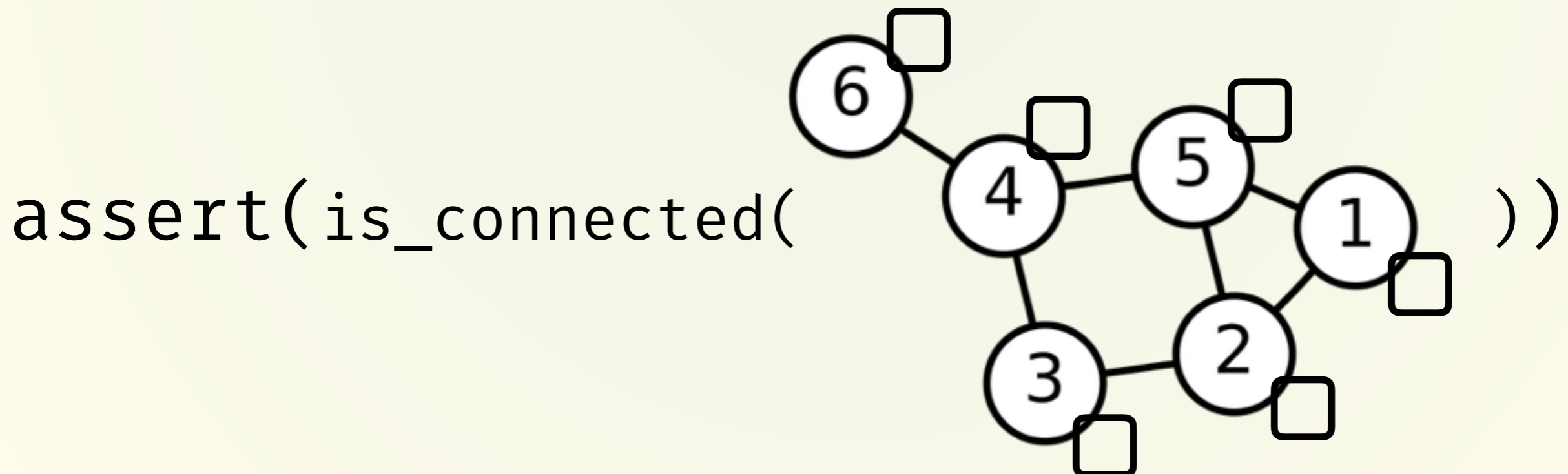
One Stone

- A programming abstraction/library
- actor based, with only local state
 - message passing only

- actor based, with only local state
- message passing only

Example: Graph connectedness:

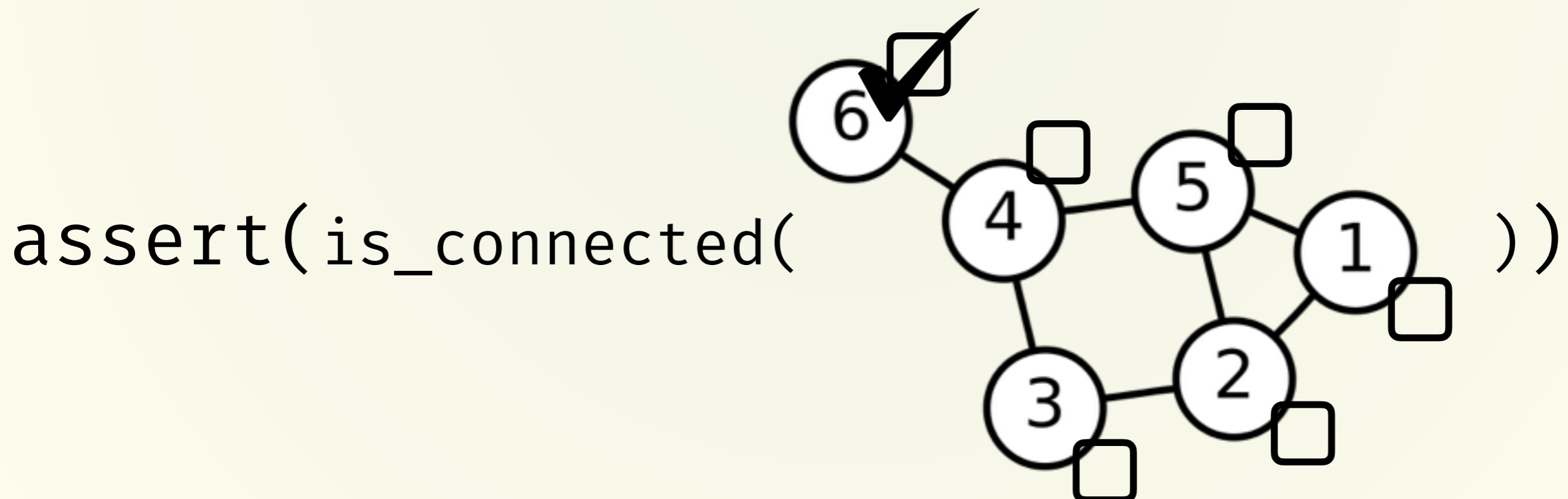
- One message: PING
- local state: a bool (initial: = false)



- actor based, with only local state
- message passing only

Example: Graph connectedness:

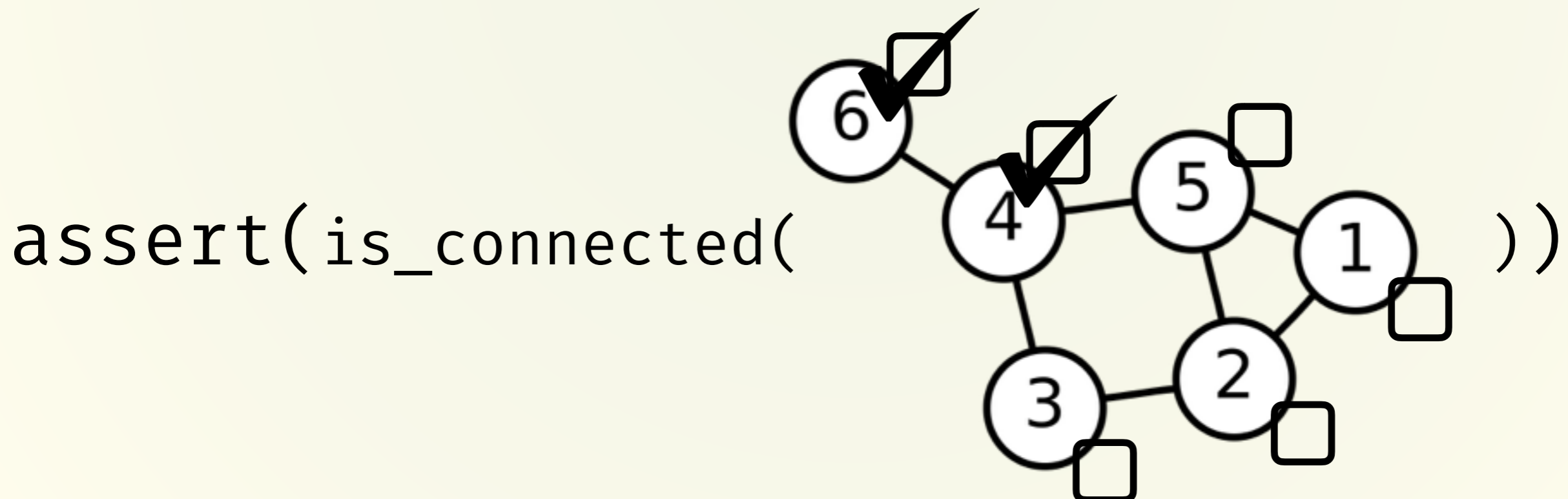
- One message: **PING**
- local state: a bool (initial: = false)



- actor based, with only local state
- message passing only

Example: Graph connectedness:

- One message: **PING**
- local state: a bool (initial: = false)

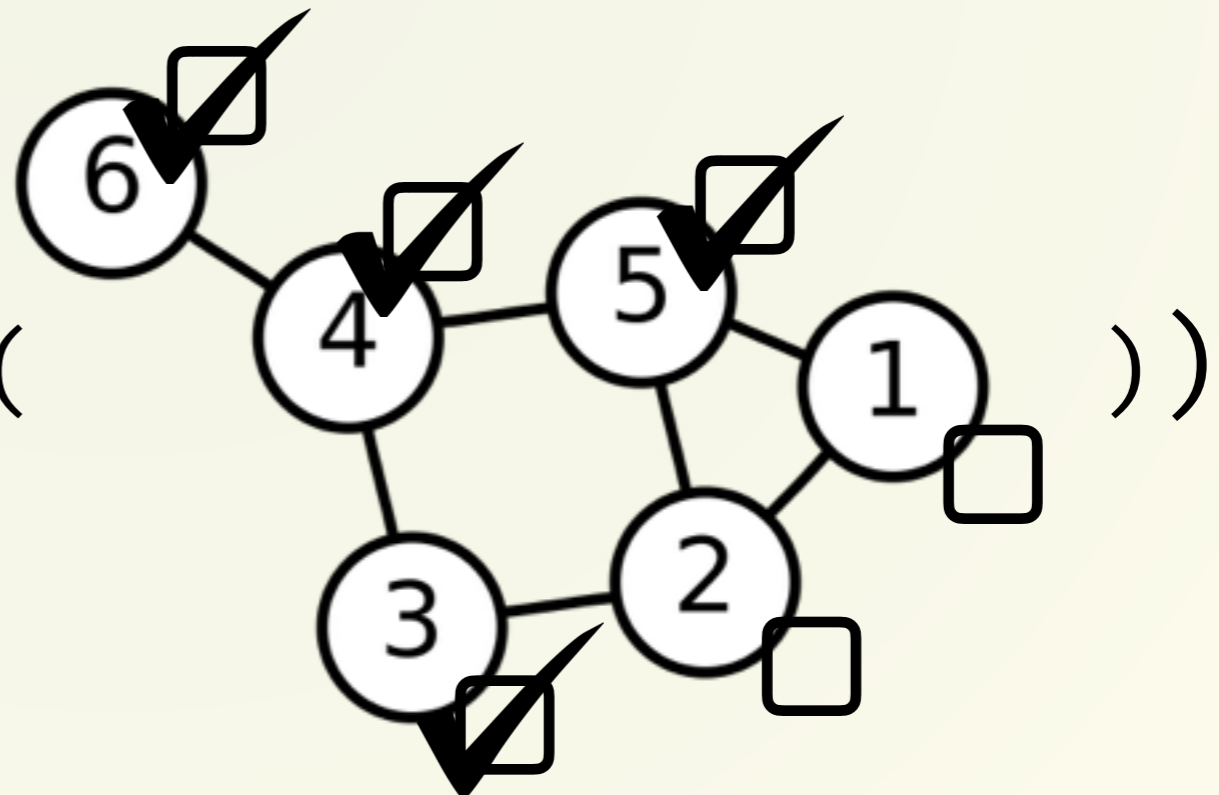


- actor based, with only local state
- message passing only

Example: Graph connectedness:

- One message: PING
- local state: a bool (initial: = false)

```
assert(is_connected(
```

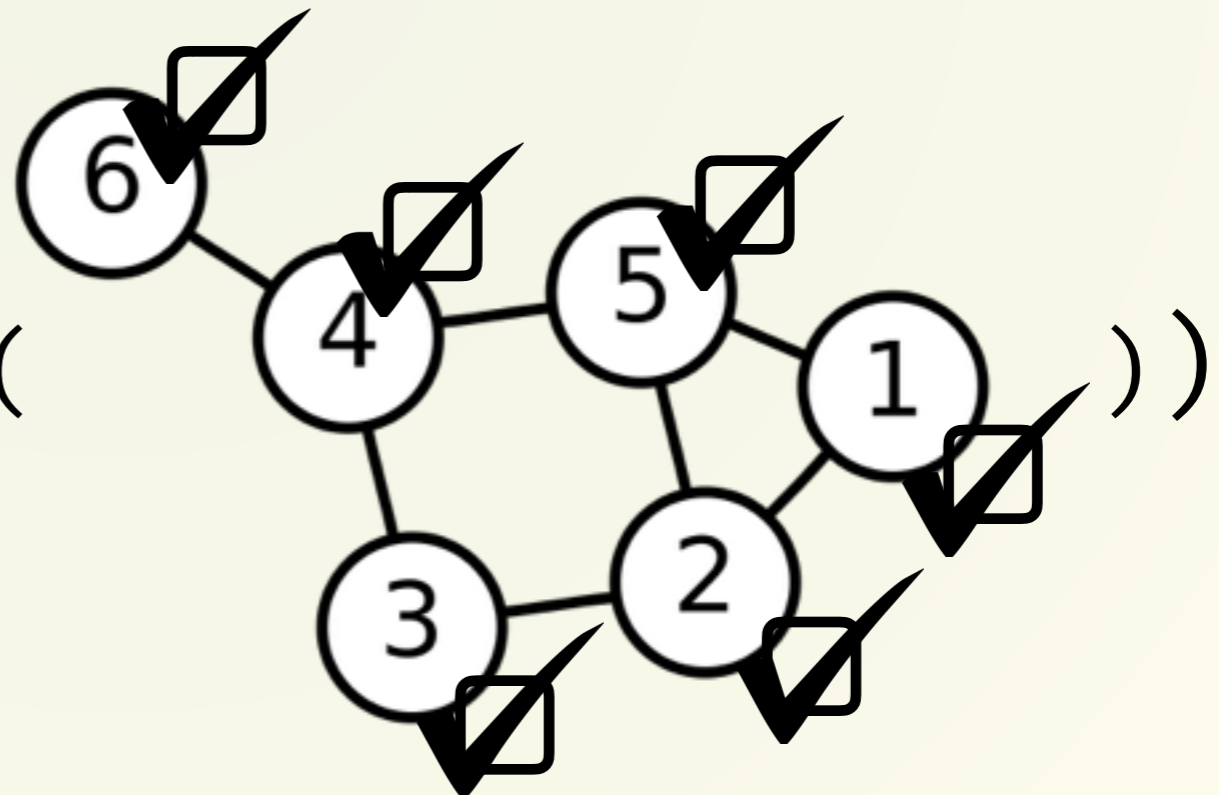


- actor based, with only local state
- message passing only

Example: Graph connectedness:

- One message: PING
- local state: a bool (initial: = false)

```
assert(is_connected(
```



- actor based, with only local state
- message passing only

Example: Graph connectedness:

- One message: **PING**
- local state: a bool (initial: = false)

`assert(`



`)`

- actor based, with only local state

⇒ **All state changes** have to go through the abstraction.

`this.f.change()` / `send(CHANGE, this.f)`

⇒ Can choose **when/how often to apply** state changes.

- message passing only

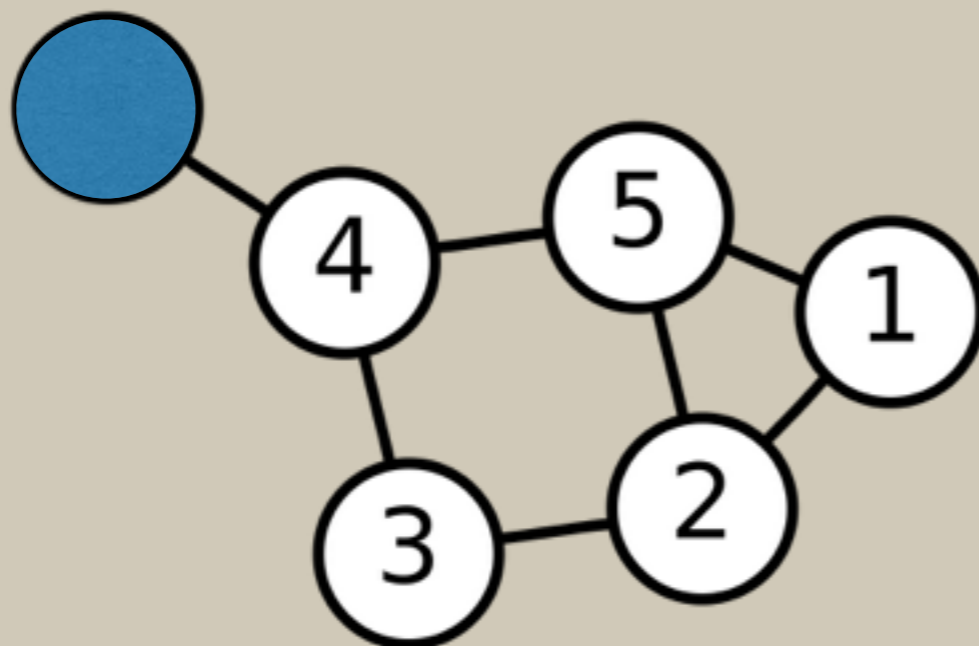
- ⇒ **All state changes** have to go through the abstraction.
- ⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
We deliver all messages in an
arbitrary, but deterministic order!



race conditions

Step 1



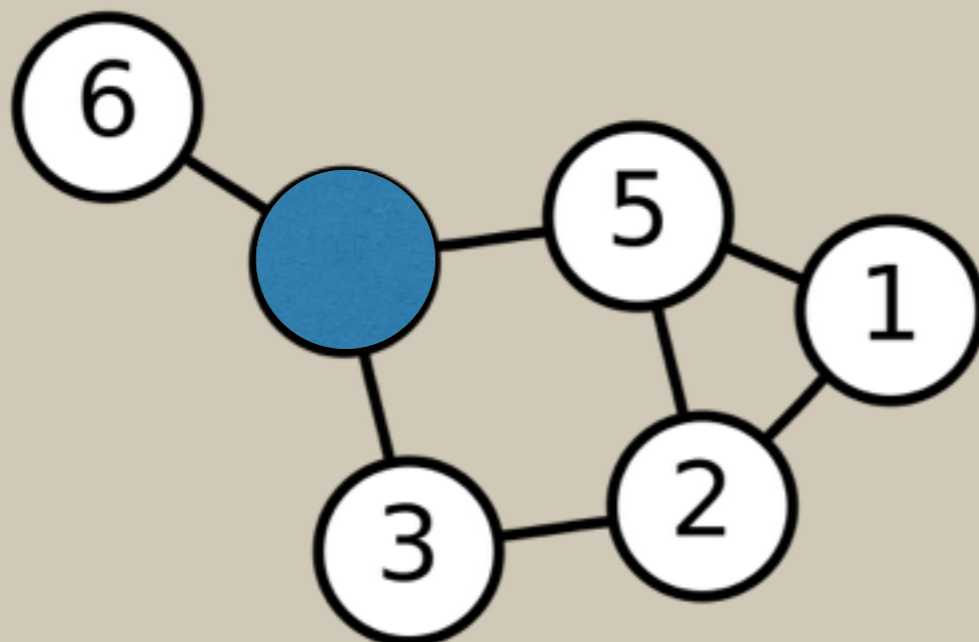
- ⇒ **All state changes** have to go through the abstraction.
- ⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
We deliver all messages in an
arbitrary, but deterministic order!



race conditions

Step 2



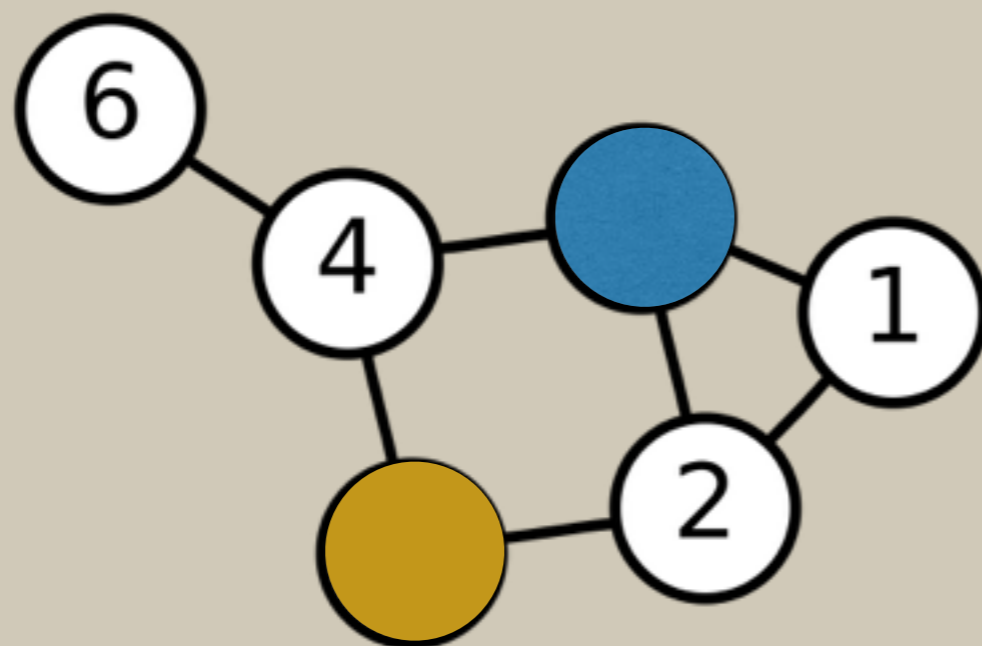
- ⇒ **All state changes** have to go through the abstraction.
- ⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
We deliver all messages in an
arbitrary, but deterministic order!



race conditions

Step 3



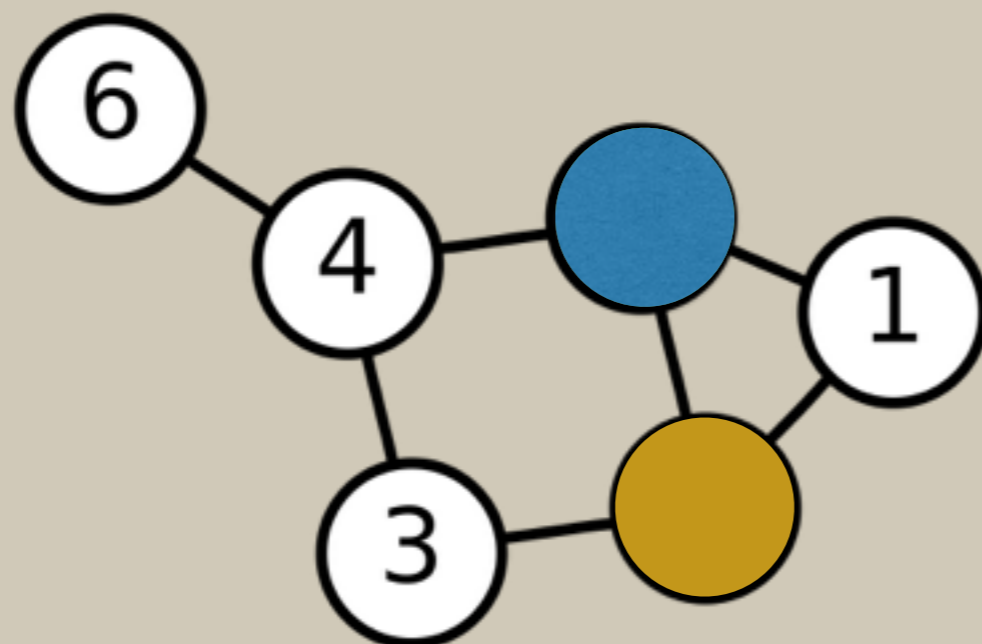
- ⇒ **All state changes** have to go through the abstraction.
- ⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and deliver
messages in **deterministic order!**



race conditions

Step 4



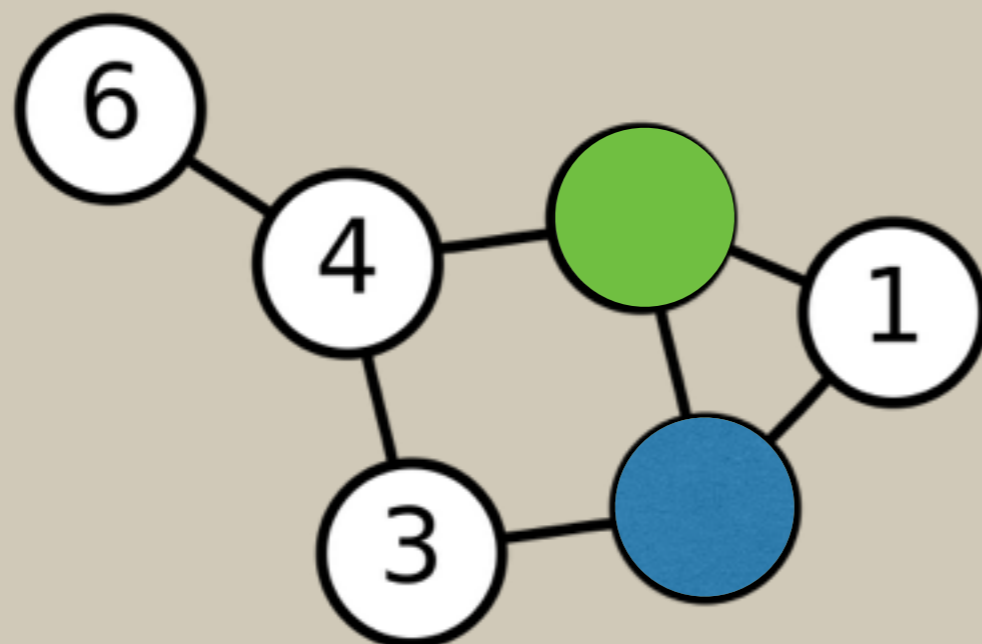
- ⇒ **All state changes** have to go through the abstraction.
- ⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and deliver
messages in **deterministic order!**



race conditions

Step 4



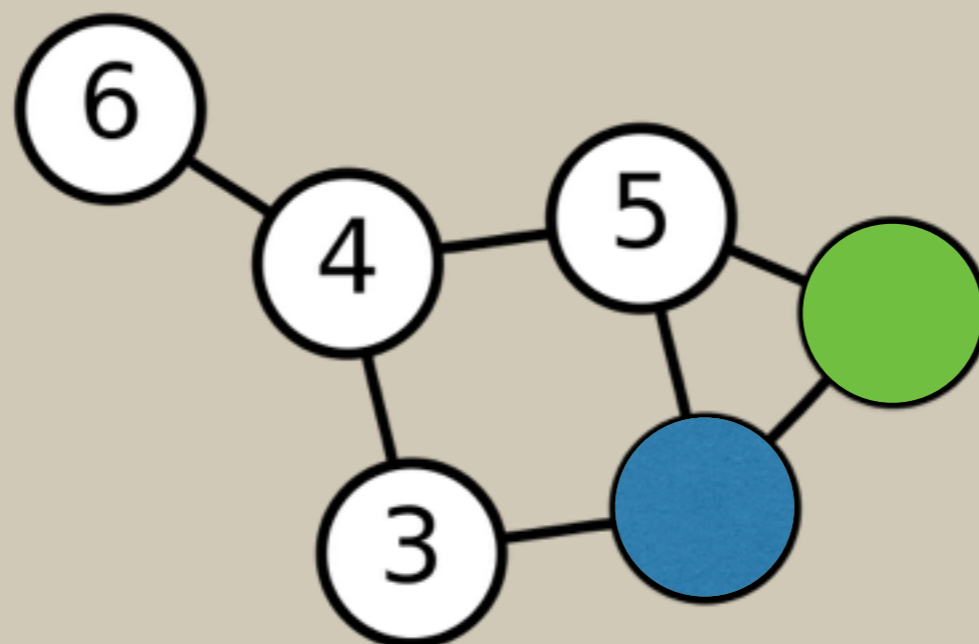
- ⇒ **All state changes** have to go through the abstraction.
- ⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and deliver
messages in **deterministic order!**



race conditions

Step 4



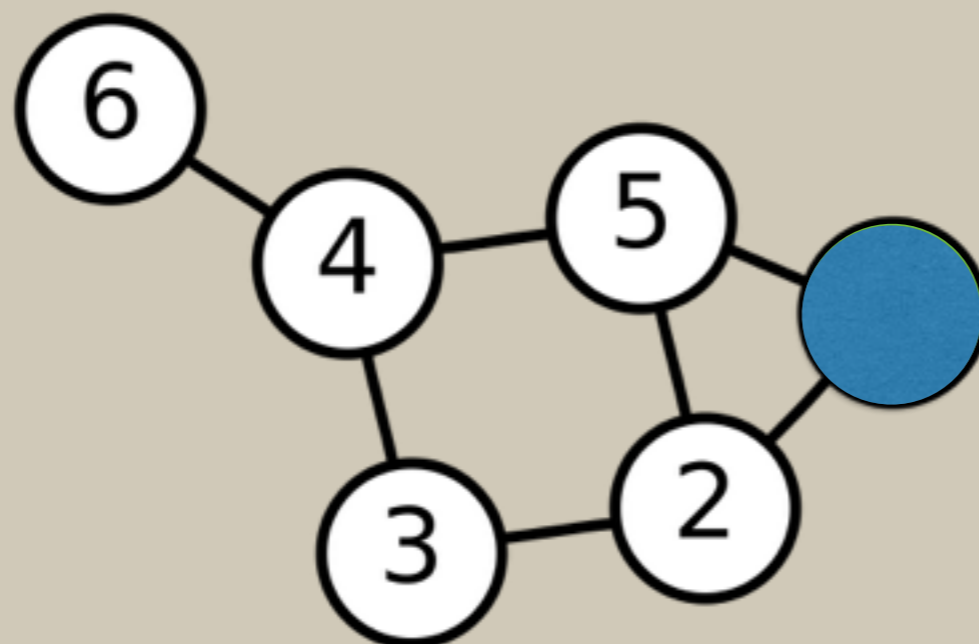
- ⇒ **All state changes** have to go through the abstraction.
- ⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and deliver
messages in **deterministic order!**



race conditions

Step 5



- ⇒ **All state changes** have to go through the abstraction.
- ⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and deliver
messages in **deterministic order!**



race conditions

- ⇒ **All state changes** have to go through the abstraction.
- ⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and deliver
messages in **deterministic order!**



race conditions



- ⇒ **All state changes** have to go through the abstraction.
- ⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and persist machine state
before each step



double espresso

“Persistent data is stored as files on a distributed storage system, GFS [19], or in Bigtable [9], and temporary data such as buffered messages on local disk.”

Invariant: only successful states
are persisted

Safety for performance tradeoff.

- ⇒ **All state changes** have to go through the abstraction.
- ⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and persist machine state
before each step



fault tolerance

“Persistent data is stored as files on a distributed storage system, GFS [19], or in Bigtable [9], and temporary data such as buffered messages on local disk.”

Invariant: only successful states
are persisted

Safety for performance tradeoff.

Slides at: <http://stbr.me/pregel-presentation>