

ROS IPC analysis

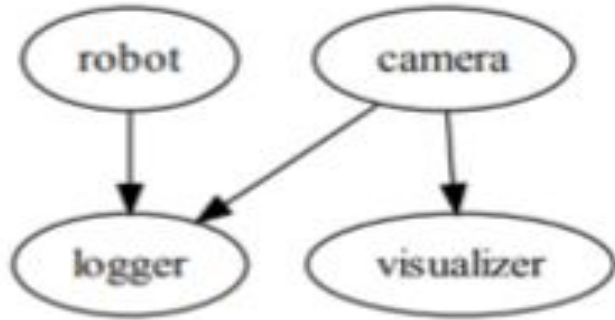
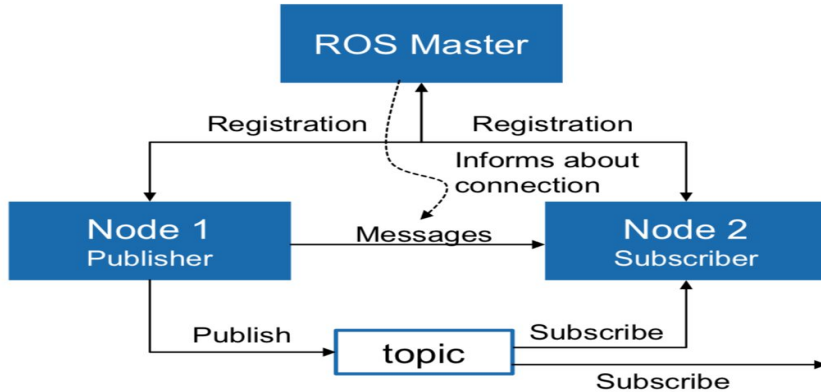
Build system and File Structure

- A build system is responsible for generating 'targets' from raw source code that can be used by an end user. These targets may be in the form of libraries, executable programs, generated scripts.
- Source code is organized into 'packages'. This is expressed in some set of configuration files read by the build system. With CMake, it is specified in a file called 'CMakeLists.txt', which is responsible for preparing and executing the build process.
- ROS utilizes a custom build system, catkin, that extends CMake to manage dependencies between packages. package.xml is used to define packaging and system build dependencies .

What is ROS?

- ROS is a meta-operating system designed for robotics software development, commonly over a cluster of computers
- Node based message passing system (subscriber / publisher)
- Support many programming languages (Python, C++)

IPC in ROS



- Nodes are processes that perform computation.Nodes communicate with each other by passing messages.
- ROS data type used when subscribing or publishing to a topic.
- Messages can be composed of other messages, and arrays of other messages, nested arbitrarily deep.
- A node sends a message by publishing it to a given topic.
- A node that is interested in a certain kind of data will subscribe to the appropriate topic.
- There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.

In general, publishers and subscribers are not aware of each others' existence

IPC in ROS

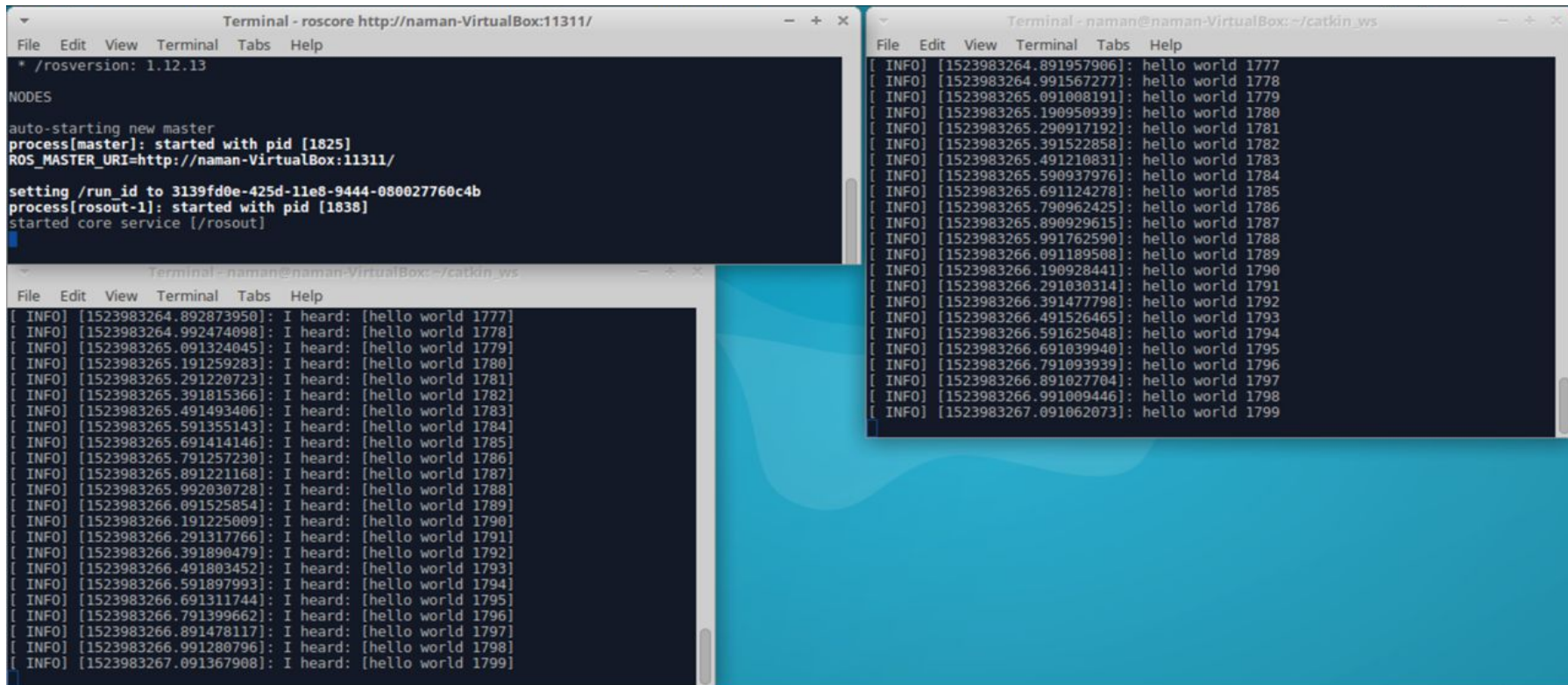
What the publisher does:

- Tells the master about the length of the publishing queue and the topic name so that the master notifies the subscribers that data is about to be published.
- A message is broadcast on ROS using a message-adapted class, generally generated from a msgfile.

What a subscriber does:

- Subscribes to a topic via the master
- When a message arrives it calls the callback function for that topic which gets the message from a pointer that is safely stored elsewhere and processes it in the way it wants to

Running Example of IPC in ROS



The image displays three terminal windows from a ROS environment, illustrating the process of starting a ROS master and nodes.

Terminal 1 (Top Left): Shows the ROS master starting. The title bar is "Terminal - roscore http://naman-VirtualBox:11311/". The output includes:

```
* /rosversion: 1.12.13
NODES
auto-starting new master
process[master]: started with pid [1825]
ROS_MASTER_URI=http://naman-VirtualBox:11311/

setting /run_id to 3139fd0e-425d-11e8-9444-080027760c4b
process[rosout-1]: started with pid [1838]
started core service [/rosout]
```

Terminal 2 (Top Right): Shows a node outputting "hello world" messages. The title bar is "Terminal - naman@naman-VirtualBox:~/catkin_ws". The output is a list of INFO messages with timestamps and IDs, each followed by "hello world" and a sequence number (e.g., 1777, 1778, etc.).

Terminal 3 (Bottom Left): Shows another node outputting "hello world" messages. The title bar is "Terminal - naman@naman-VirtualBox:~/catkin_ws". The output is a list of INFO messages with timestamps and IDs, each followed by "I heard: [hello world" and a sequence number (e.g., 1777, 1778, etc.).

ROS2 vs ROS1

ROS1:

Only support a single robot

No real-time requirements

ROS2:

Support multi-robot systems

Support for real-time control

Remove gap between prototype and product

IPC in ROS

- ROS message is a data structure that defines the type of message being sent through topic.
- ROS message is like JSON, which contains nested structure of objects and any primitive types.
- It is defined in *.msg file

**.msg* Message definition

```
int number
double width
string description
etc.
```


Current issues in ROS Nodelet

In ROS, Nodelet is used to create multiple Nodes on a single process, which result in several issues:

- Nodelet and Node have different API, which increases developer's burden.
- Creation and running Nodelet are complicated.
- User locking causes starvation

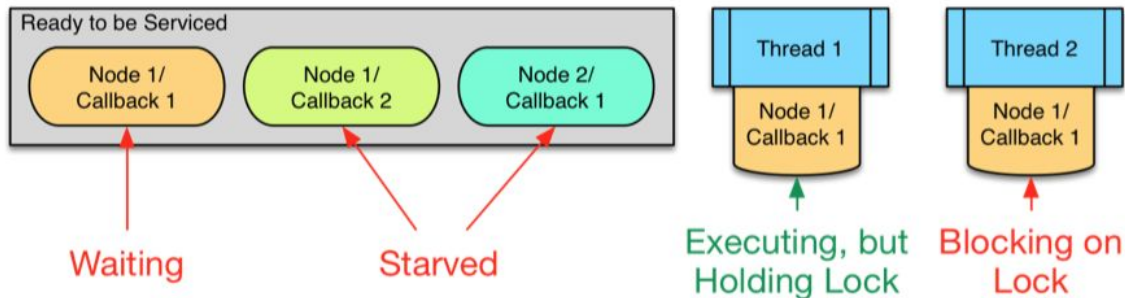
Due to the fact that it is not possible to create multiple nodes on a single process, Nodelet is necessary when multiple algorithms need to be run on single process, which leads to above issues.

Thread pool starvation

A nodelet manager has a pool of threads which is shared across all nodelets run within the manager.

If nodelets are blocking threads they may prevent other nodelets from getting callbacks.

Note: Even the single threaded Node Handles can consume 1 thread of the pool per nodelet.



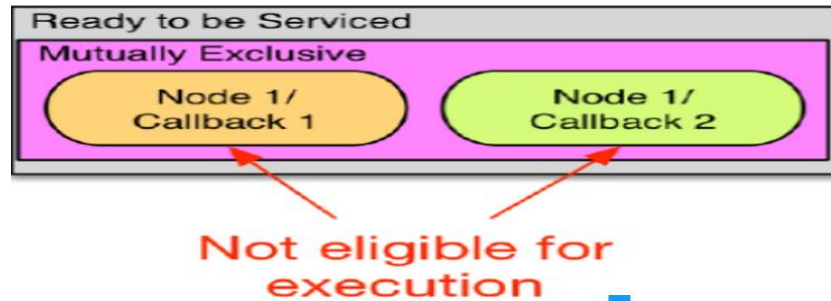
Starvation solution

One solution is to avoid using locks but split callbacks into groups:

- Runnable callback groups are callbacks that are able to run simultaneously with other threads and callbacks in the same or other groups
- Exclusive callback groups are callbacks can only run simultaneously with callbacks in other groups.

User defines which Node / Callback combination belongs to which group.

This mechanism should solve starvation in most of the scenarios since which thread and callback get to run completely depends on user's choice of group.



Node naming issues in ROS

ROS 1 does not allow duplicate Node name by shutting down existing Node with the name.

- Unnecessary since subscriber and publisher are anonymous
- Increase the burden on developer for better naming.

Naming solution

Idea: Add unique id to each node when initiating the node in `rosegaph/xmlrpc.py`. Keep the unique ids that have been used in list. When node is to be shutdown, remove the unique id from the list.

Associate file changes: `/rosgaph/names.py`, `/rosgaph/xmlrpc.py`,
`/rosmaster/master.py`, `/rosmaster/registrations.py`

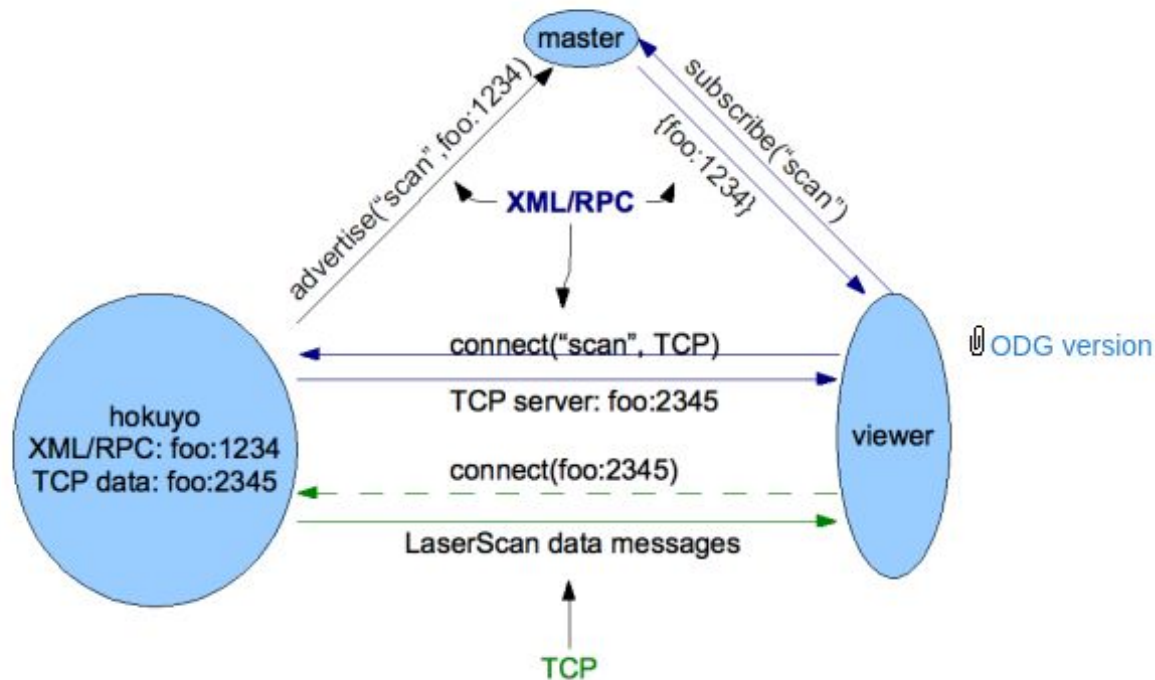
*ROS2 not yet implemented solution to this issue.

Pub/sub issues in ROS

- When a node wants to publish to a certain “topic”, it asks a central node, or the **master node** for the network addresses (IP and port) of the subscribed nodes, and sets up connections with those nodes. This creates a single point of failure- if master node goes down, nodes cannot discover and communicate with each other.
- Furthermore, **master node** does not scale well- it is currently a single process, which might be overloaded with requests if there are many nodes trying to communicate with each other.
- The master node also has a **Parameter Server**, a key value store that nodes can query. This is not scalable either.
- Want to make changes to existing ROS 1.x

Pub/sub in ROS

0.1 Example



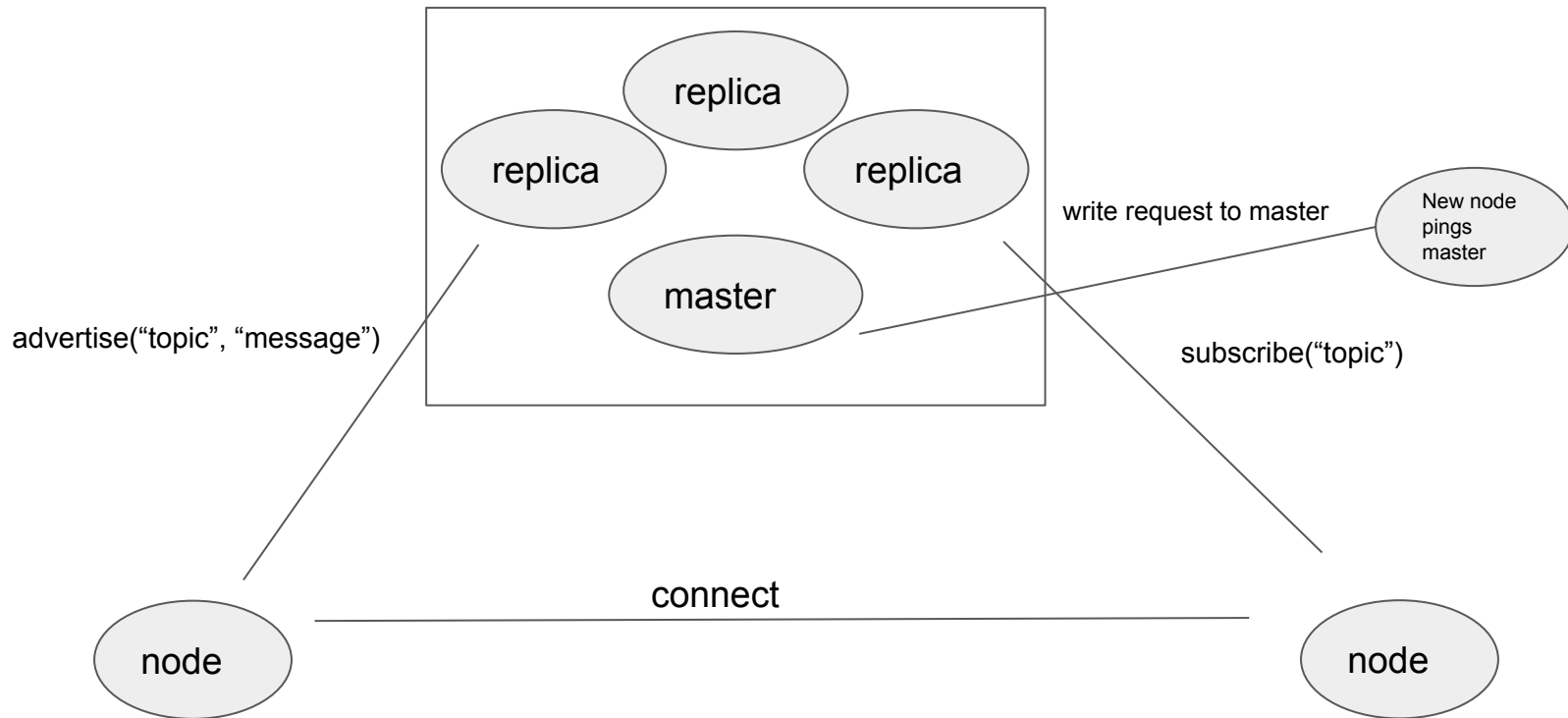
Replicating Master Node

- Replicate master node with some consensus algorithm (Raft, Paxos, etc.)
- 2 types of requests to master node- write and read.
 - **Write:** When master node discovers a node has gone down or come up, it must write this new information into its database, and replicate to its replicas. Only the master replica can do this.
 - **Read:** When a node wants to communicate with another, it issues read request to master. Constitutes vast majority of requests. Both master replica and its other replicas may respond to these requests.

Replicated master node in ROS

Replicas may execute read requests, only master can execute write requests.

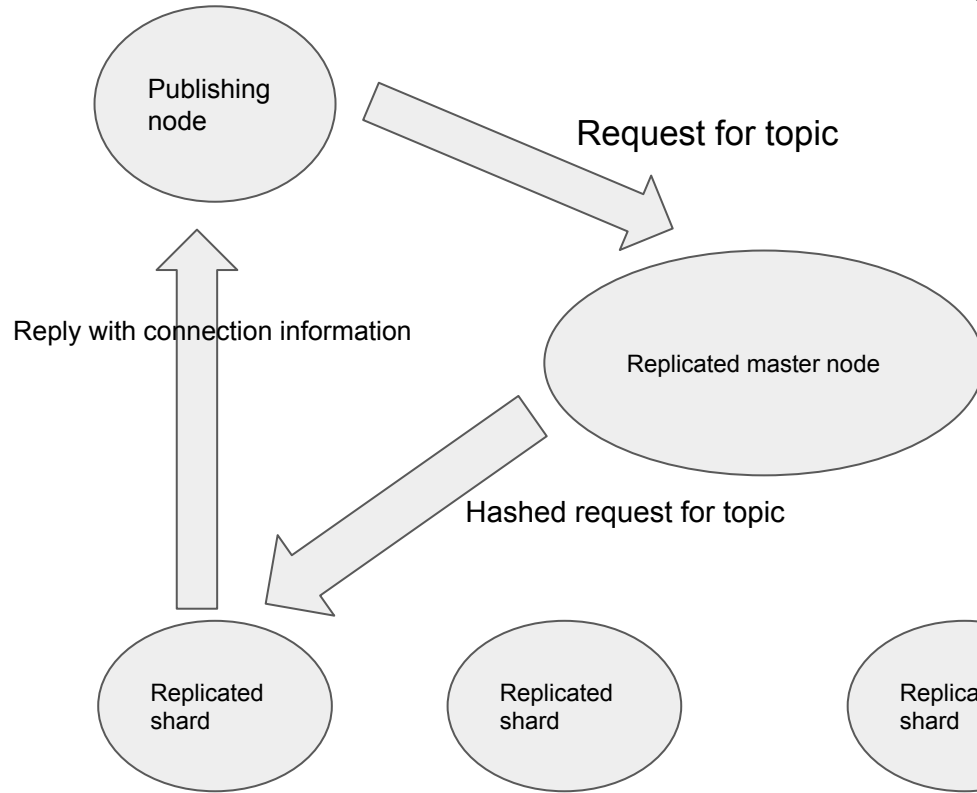
While master down and election is happening, replicas may still accept pub/sub requests.



Improving master node for scalability

- We've accounted for the reliability problem.
- What if there is a large number of nodes in a cluster? Master node can't serve all requests with reasonable latency.
- Solution: **Multiple master nodes!**

Sharding master node

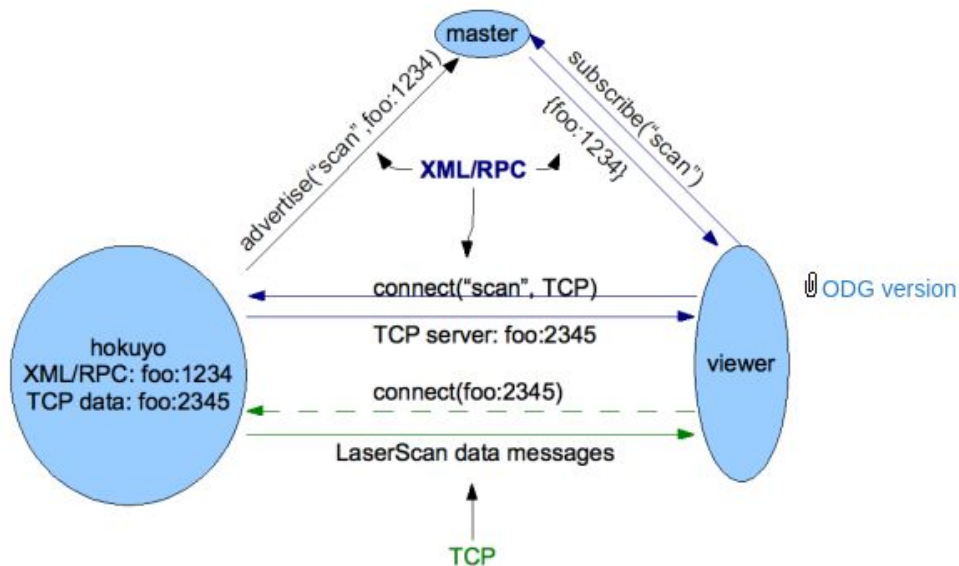


1. Node wants to publish, requests connection information of subscribers.
2. Master node now acts like a reverse proxy, delegating requests to shards by hashing them.
3. Shard replies to the node with list of subscribers.
4. Node publishes to subscribers as usual.

Sharding master node

From the node's point of view, everything still looks the same.

0.1 Example



Sharding master node

What if one of the shards fails? What if a shard recovers from failure?

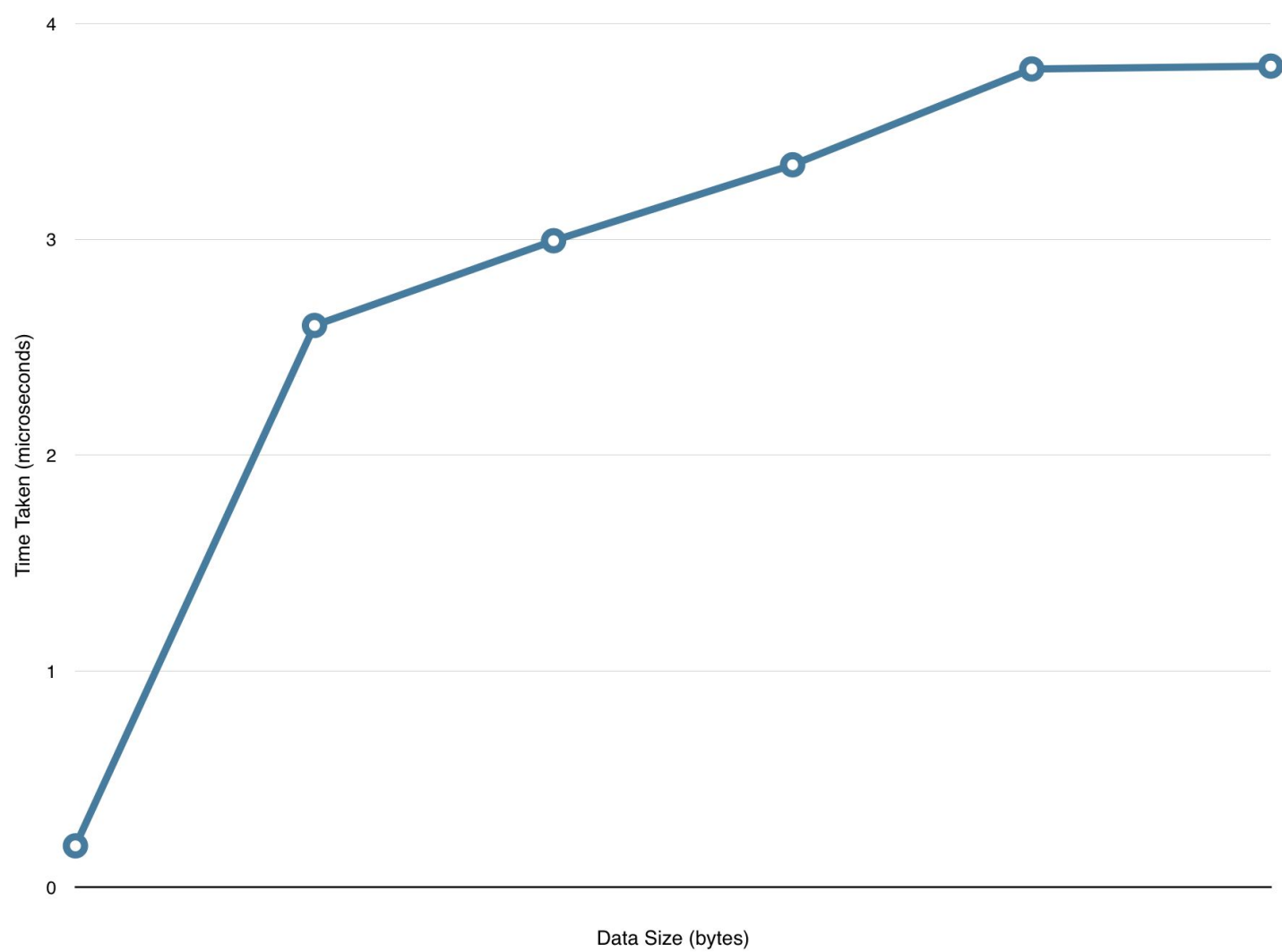
- Use consistent hashing algorithm, so that only the shard's keys get rehashed to another server, instead of the entire key space.
- Consistent hashing algorithm will not be described in detail here.

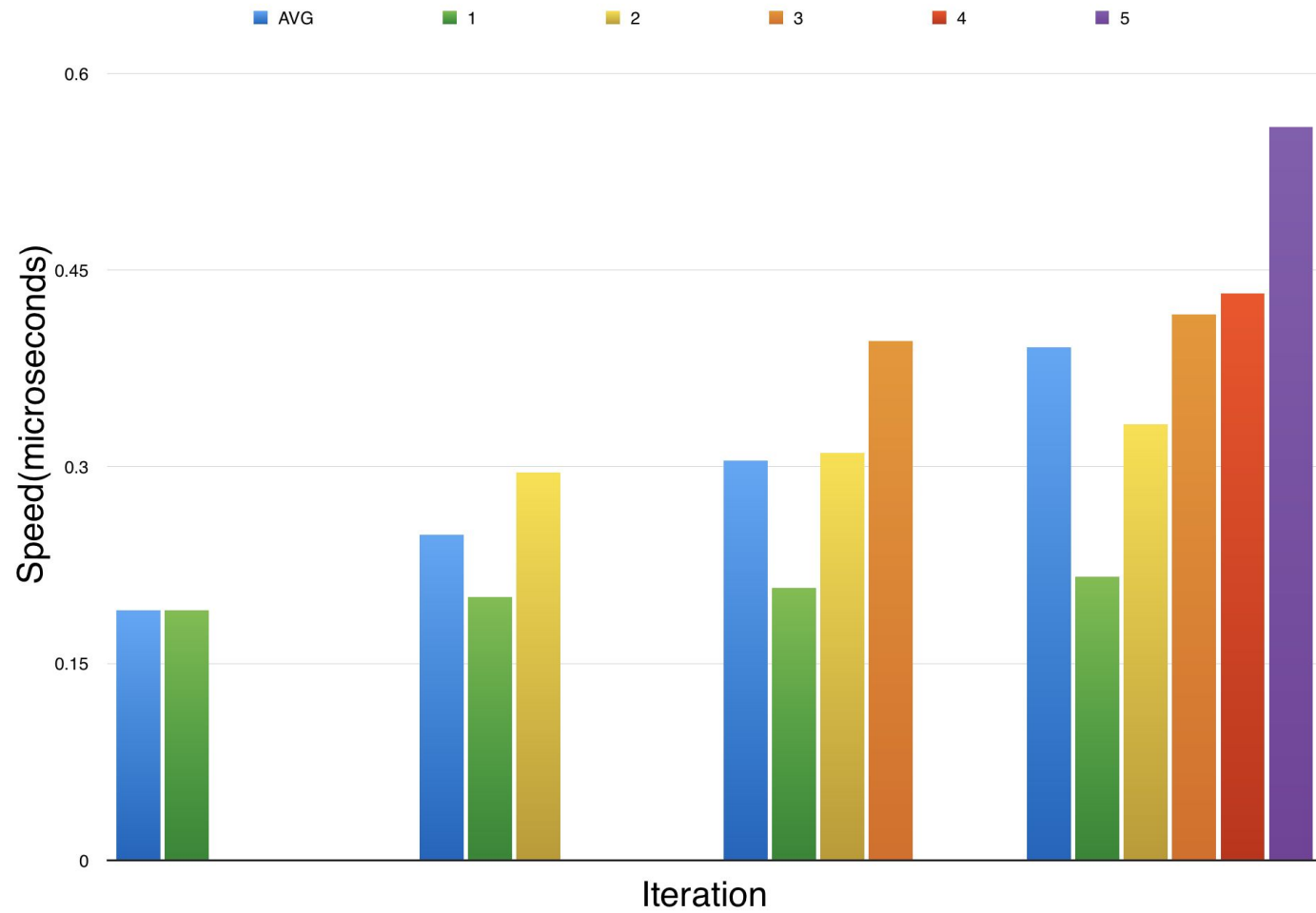
Implementation Details

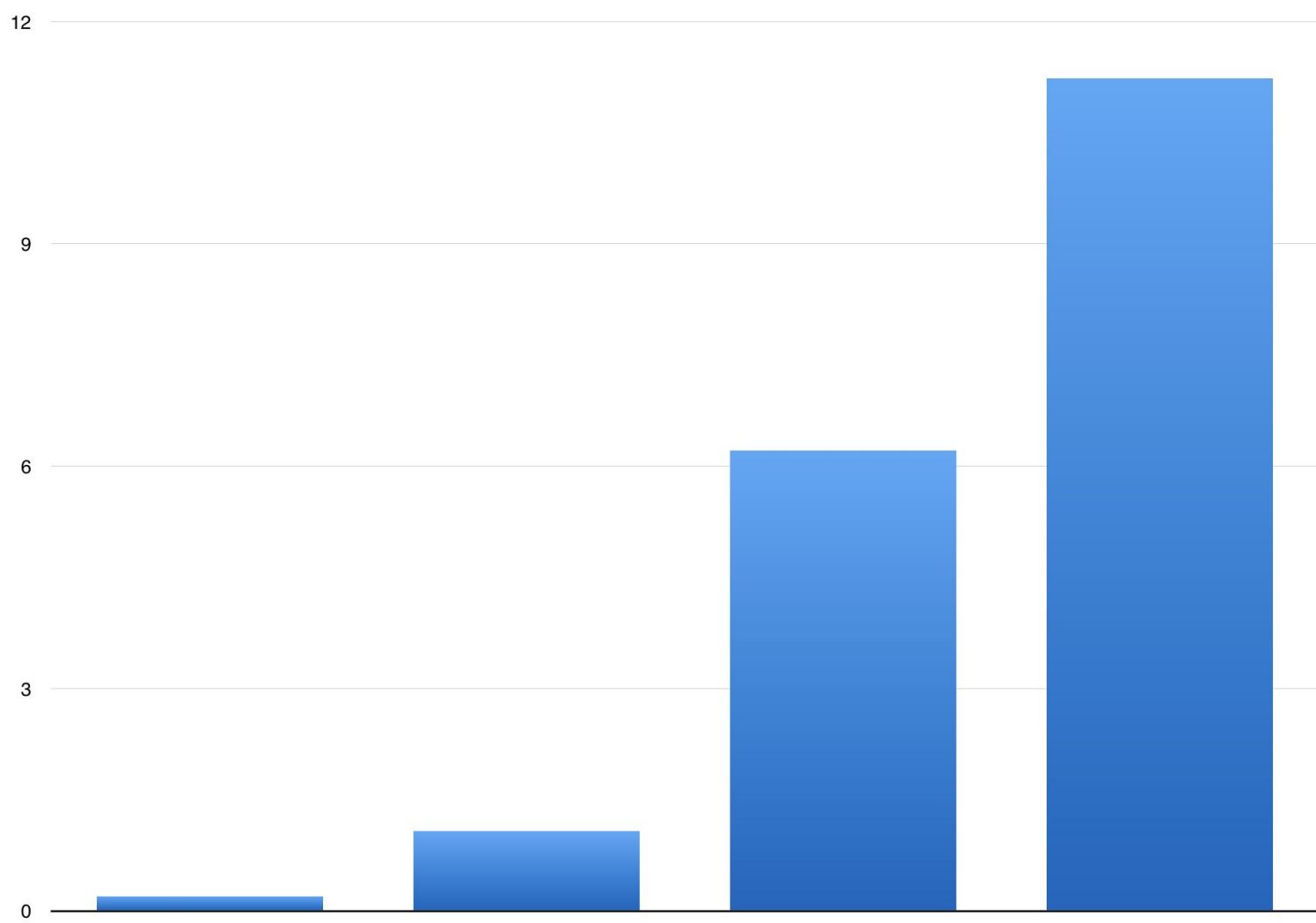
- Each node is a Master object.
- The master replica contains a table of all available shards. Master replica sends heartbeat to each shard to make sure they are continuously available.
- All requests arrive at master replica- same well-known network address as the old master node.
- The shard itself, not the master replica, responds to the node's request.
- As a result, shard must know network address of node, which is passed down through metadata by master replica.
- All of these nodes communicate through ROS Services (request/reply paradigm).

Benchmarking with ROS

- Publisher publishing 100 timestamps per second
- Subscriber(s) receiving each timestamp and immediately calculating the difference
- How does this change as the size of the message changes?
- How does this change as we have more subscribers?
- How does this change as we have more publishers?







Observations

- Extremely low data loss, even for large payloads
- No ordering guarantee for many to one, one to many models. Information can arrive in any order
- High serialization cost, but still better than memory mapping