MET CS 622: Batch and Stream Processing

Reza Rawassizadeh

Outline

- Batch Processing
- Stream Processing
  - Asynchronous Messaging
  - Synchronous Messaging

Outline

- **Batch Processing**
- Stream Processing
  - Asynchronous Messaging
  - Synchronous Messaging

Batch Processing (Offline systems)

- Sometimes the processing task we define for our application is large and it is not possible to get it done immediately in real-time or near real-time.

- Especially in web applications and with the advent of HTTP based web services such as REST, some tasks should be given enough time to compile.

- Batch processing takes a large amount of input data and uses a background job to process them.

Stream Processing (near real-time systems)

- Stream processing is something between real-time and batch processing.

- Some times it is called near real-time processing as well.

Batch Processing

- To my knowledge, there is no commercialized software application at an industrial level that does not have the batch processing.

- A very common example of batch processing is backup systems.

- MapReduce, the algorithm which makes Google very scalable is an example of a batch processing algorithm.

- Usually, batch processing jobs were written with OS scripting languages, such as Unix Scripts.

- It is very common that batch processing jobs were executed at the same machine where the data is located.

Some Example of Batch processing

- **Backup daily/weekly data** into another safe storage.
- **Compressing logs and history of transactions** of the system.
- **Sorting the entire dataset** after its changes is a cumbersome and memory in-efficient task. It is better to create a batch job for sorting.
- **Clustering** data and identifying some groups in systems with heavy load could be also done in batch job.
- **Bulk database updates**, as contrasted to interactive online transaction processing (OLTP) applications.
- **Converting files format** to another format. For example, a batch job may convert proprietary and legacy files to common standard formats for end-user queries and display.
- **Image processing on large number of images**, e.g. scale and resize images.

Output of Batch processing

- The output of a batch job usually is not a report for human. It is something that is used by the application (e.g. MapReduce output is a database).

- Google used MapReduce to build indexes for its search engine.

- Now google moved to another architecture, *Cloud DataFlo*w, which is not the topic of our discussion. Nevertheless, there are still many tasks that could be done with MapReduce and you need to learn its concepts.

MapReduce

- MapReduce is a programming model that allows us to distribute our computation for a specific task among several machines.

- *It is not easily possible to assemble heterogeneous machines to perform a single resource-consuming job*.

- The challenge is to identify how can we divide large tasks across multiple machines.

- MapReduce split petabytes of data into smaller chunks and distributes those chunks in parallel among our machines (server machines).

- MapReduce is similar to OS scripts but can parallelize computation across different machines, without the need to develop complex codes for handling network communication.

- If there are events that terminate the task, MapReduce can redo the task, until all tasks were done and then it finishes the job. It means that it is a good framework to be used on unreliable systems.

MapReduce

- It has three stages, **Map**, **Shuffle** and **Reduce.**

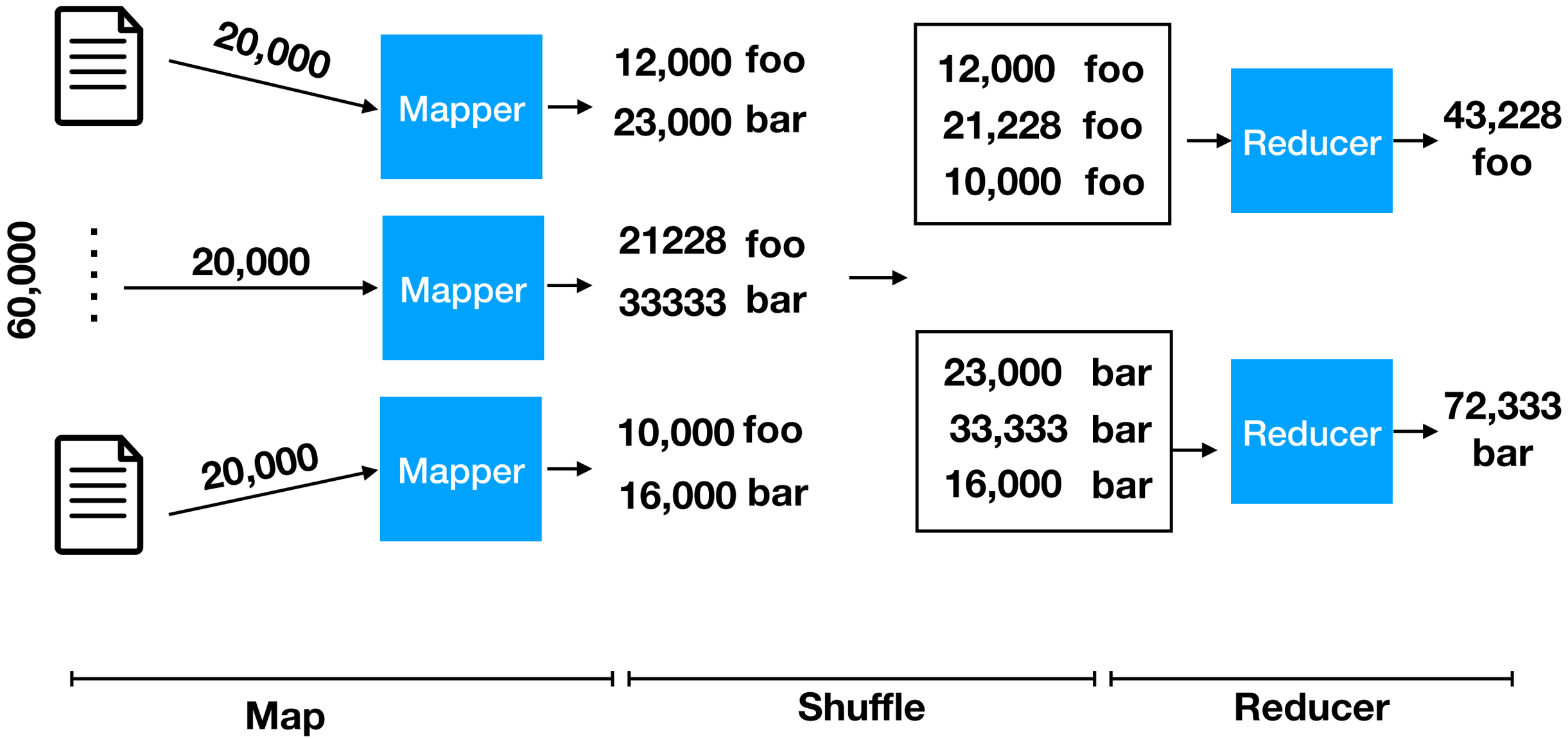
1. A set of input files will be read and will be broken into records.

2. The mapper function will be called. The role of this function is to extract key-value pairs from each input record.

3. Then all key-value pairs will be sorted.

4. The reducer will iterate over the sorted key-value pairs. If there are multiple occurrences of a key, the sorter (step 3) makes them adjacent and the reducer merges them.

Map Reduce Example



MapReduce Challenges

- The **MapReduce reads the entire content of file** (Full Table Scan). Therefore, in cases we are interested in a part of the data, using MapReduce will be unnecessary expensive.

- The **range of the program we can solve with MapReduce is very limited**. Usually it is used to search a keyword in a document, sort a database, or search for something inside a larger dataset.

- MapReduce jobs only get completed when **all mappers and reducers have been completed**. Therefore, the system **waits until the slowest machine finishes its job**.

- MapReduce is **not good for iterative algorithms**. Therefore, several machine learning algorithms that requires iteration can not benefit from MapReduce.

SPARK

- MapReduce and similar products are called *Data Flow Engines*.

- SPARK is the newer batch processing tool, which tries to resolve issues existing in MapReduce (e.g. working with graph data, iterative algorithms, etc.).

- SPARK is a clustering framework build on top of the MapReduce and extends the MapReduce framework.

SPARK

- Partitioning and fault tolerant are two major challenges of distributed systems. Let's call these systems **cluster management** systems.

- Industries are heavily using apache tools for their big data, especially Apache Hadoop framework.

- Hadoop is just one way to implement SPARK cluster management.

- The main feature of SPARK is its **in-memory cluster computing** which decrease the latency in a batch job.

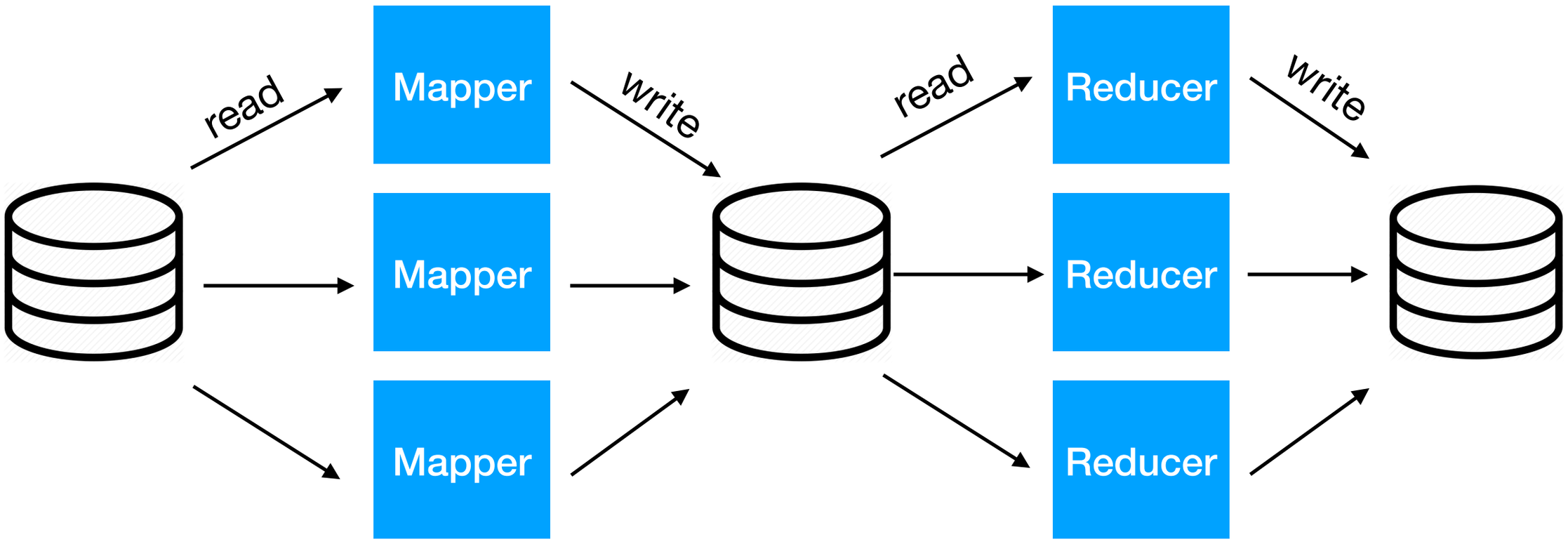
SPARK Advantages

- SPARK stores intermediate process in memory. This makes the application to run in Hadoop cluster significantly faster, not just applications in memory but also applications on disk.

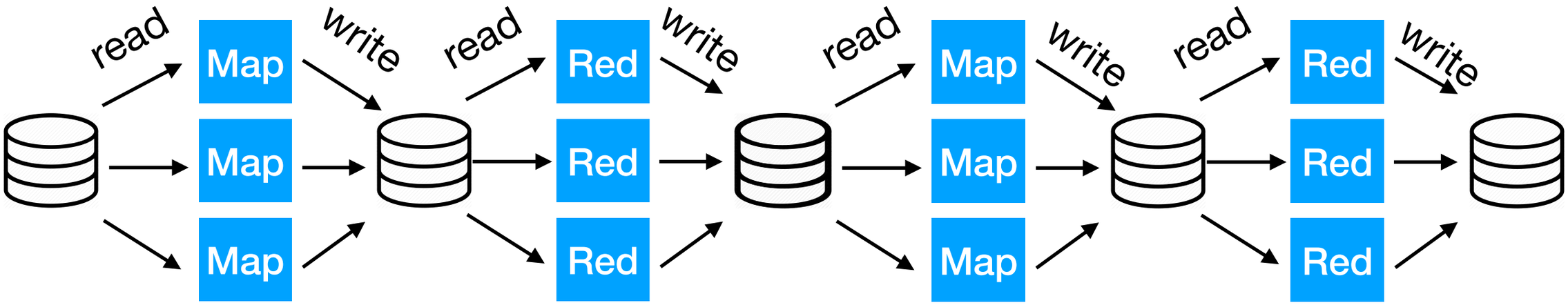
- In addition to MapReduce SPRAK supports stream data, machine learning and SQL queries as well.

- SPARK is using lots of RAM, thus it is running very fast.

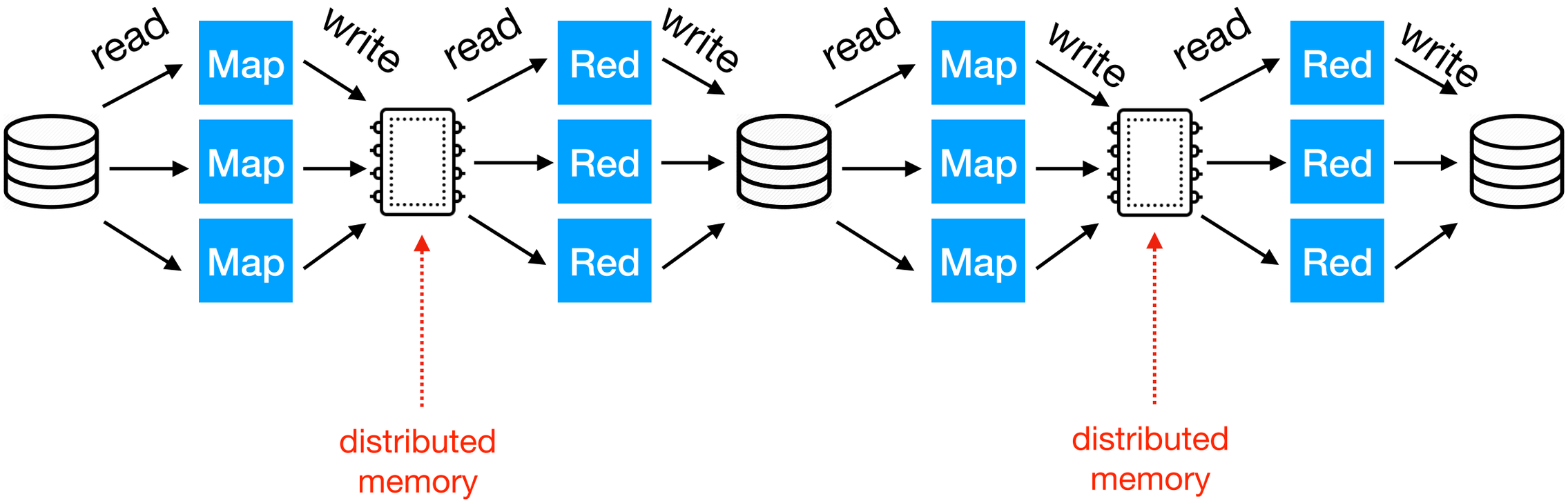
MapReduce I/O



Iterative Operation on MapReduce



Iterative Operation on SPARK RDD



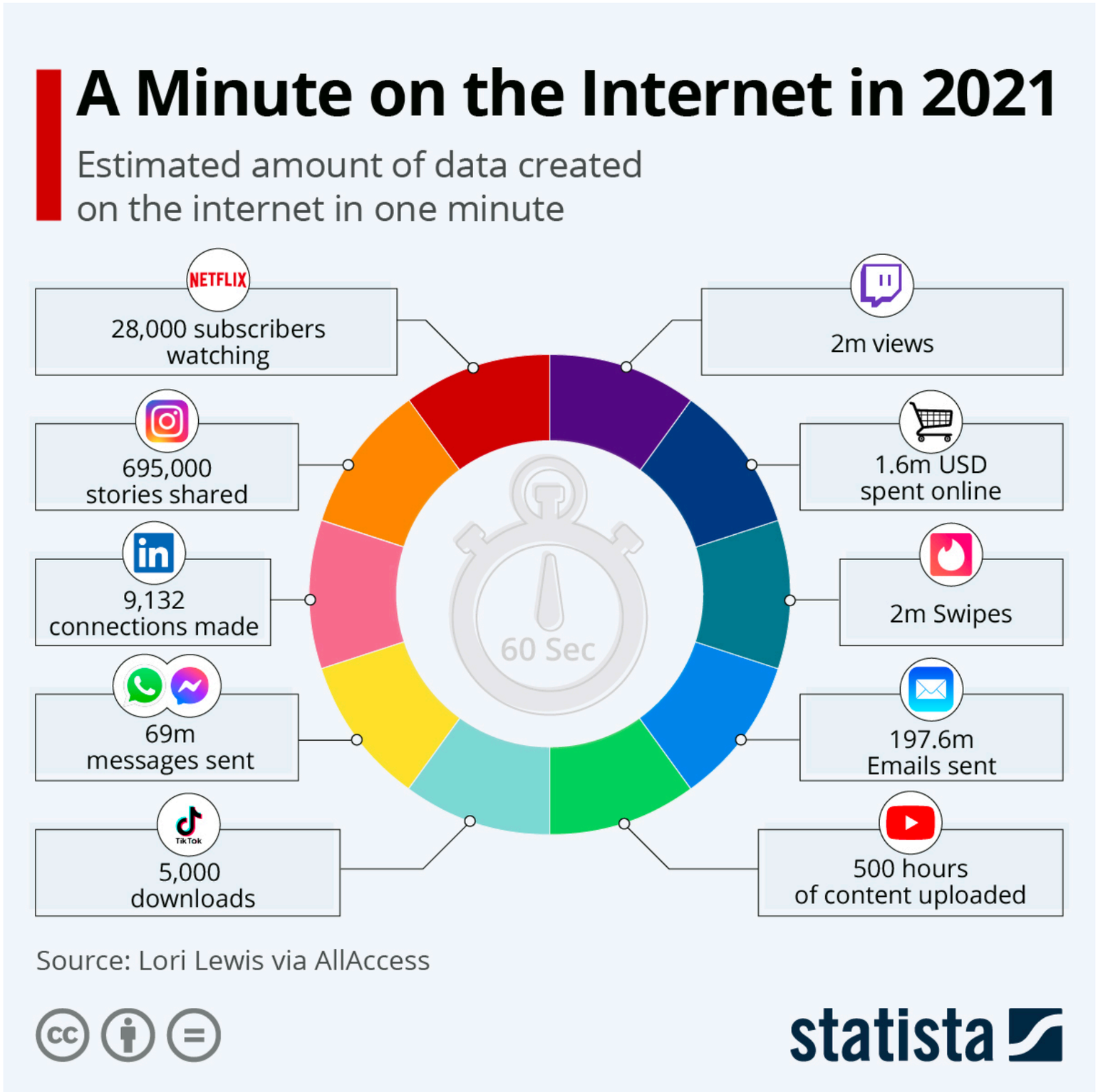
Fault Tolerance in Batch Jobs

- It is fairly easy to implement the fault tolerant in Batch job. If a job failed simply batch processor restart that job.
- Nevertheless, some times the job is getting too big. To handle big jobs, we can create small partitions called **microbatches**. More about microbatches will be explain later.

Outline

- Batch Processing
- **Stream Processing**
  - Asynchronous Messaging
  - Synchronous Messaging

Stream Processing



What is Stream Data?

- Lots of data are unbounded. They arrive gradually over time, the system logs data yesterday, last week, … and will produce logs every second.

- Some times, the changes in the data is important and we need systems to capture these changes rapidly.

- **Stream data is a data that is incrementally made available over time [Klepman ’18].**

- The Batch processing consider data as chunks of fixed length static data.

- Stream Processing is sometimes considered as opposite of Batch Processing (Not everybody agree with this definition).

What is Stream Data?

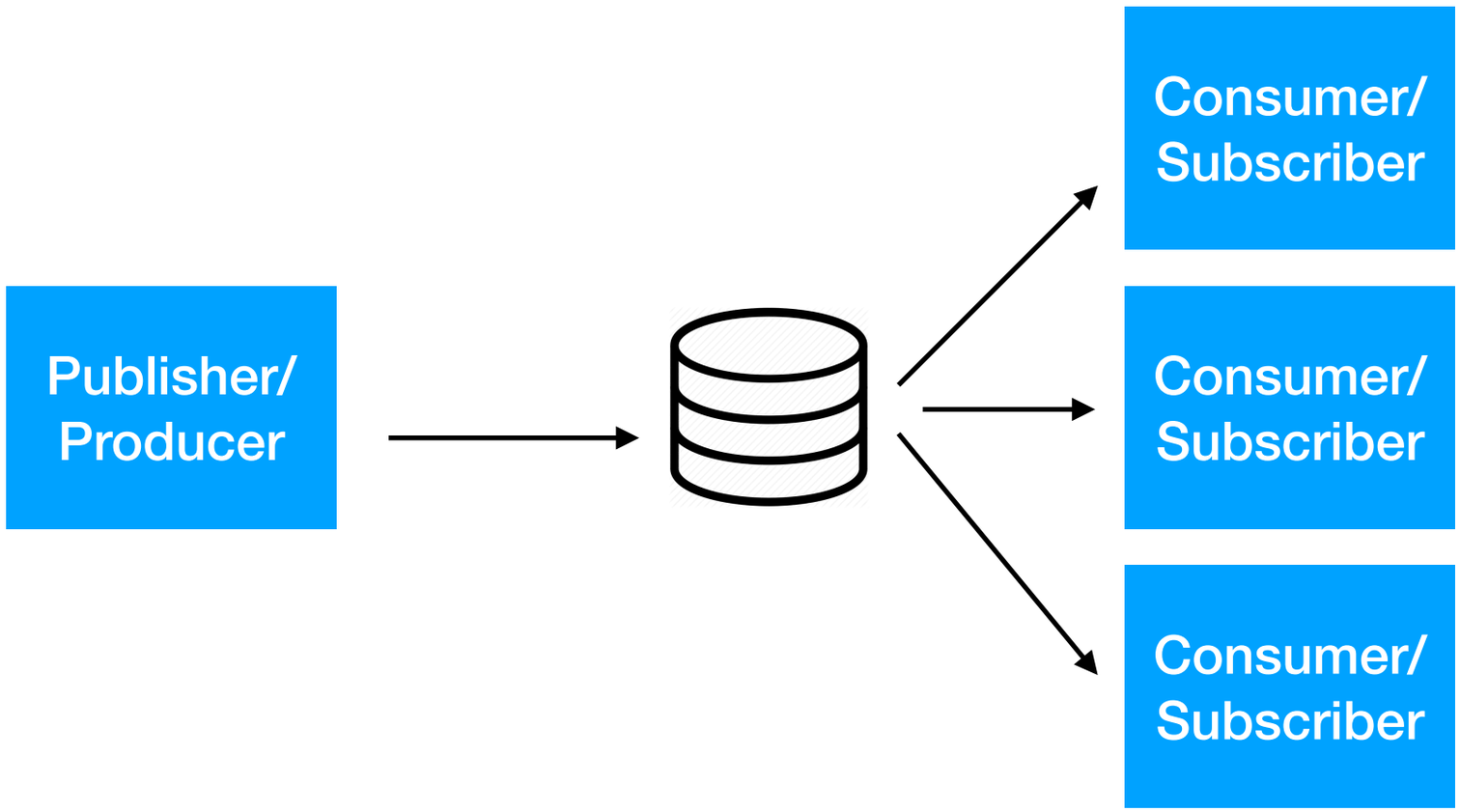
- Usually the first step in batch processing is to convert the target data file into a set of records.

- In stream processing we convert data into a set of ***events***. (event ~ record).

- In stream processing events are created by one single **producer (publisher)**, but in batch processing a data can created by multiple **jobs.**

- A periodic measurement of a sensor, temperature of a device, new logins into a social media, … are all example of data streams.

Publisher and Subscriber Model



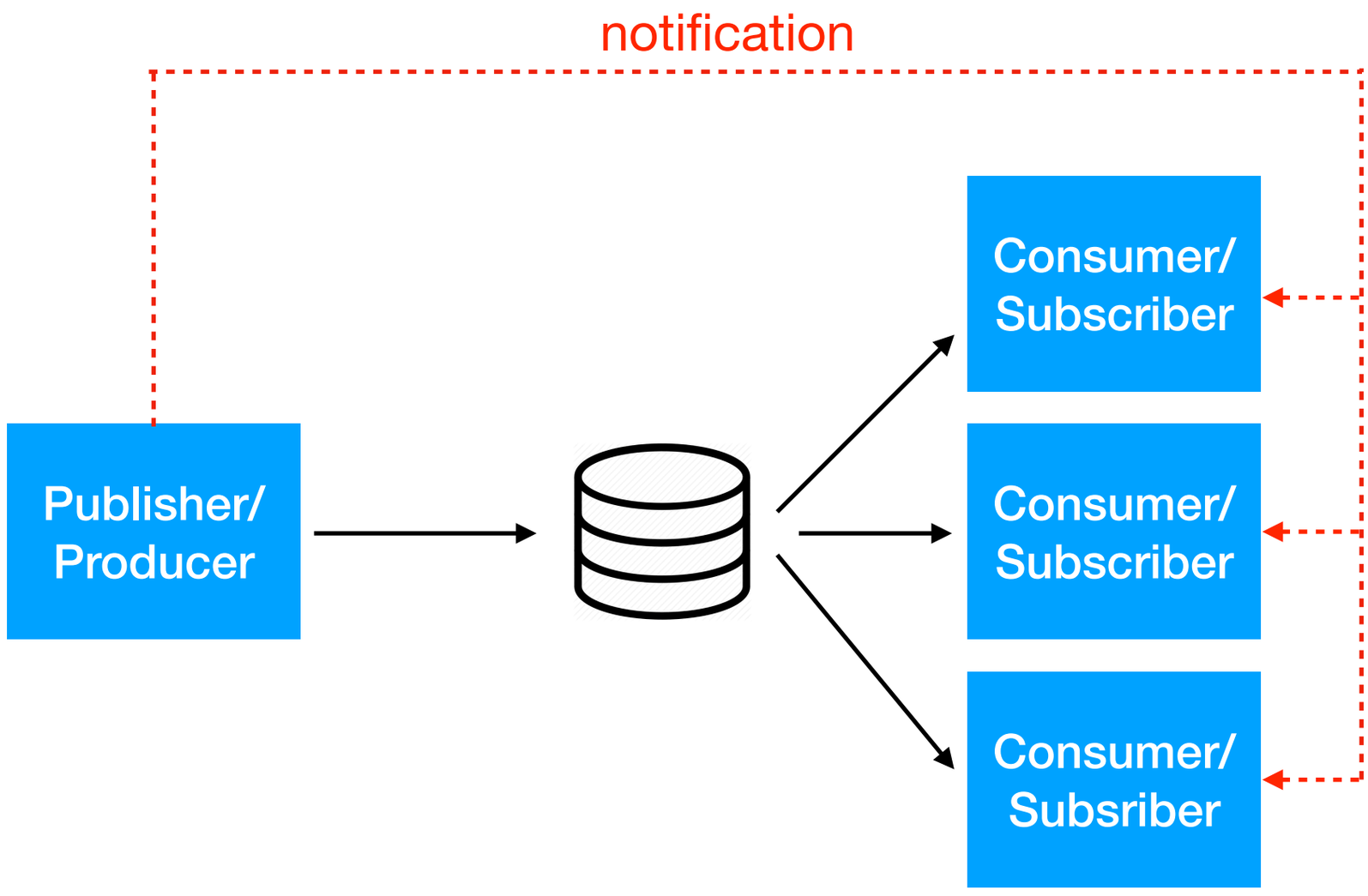
Streams

- Publisher produces the data and subscriber consumes the data.

- Polling information from a dataset is an expensive process and it is usually associated with I/O operation.

- One way to reduce the cost of polling is to avoid frequent polling, by enabling the publisher (data producer) to notify the subscriber (data consumer). In other words, the subscriber does not need to do it frequently, whenever the information is available the publisher notifies the subscriber.

Resolving the Frequent Polling Issue



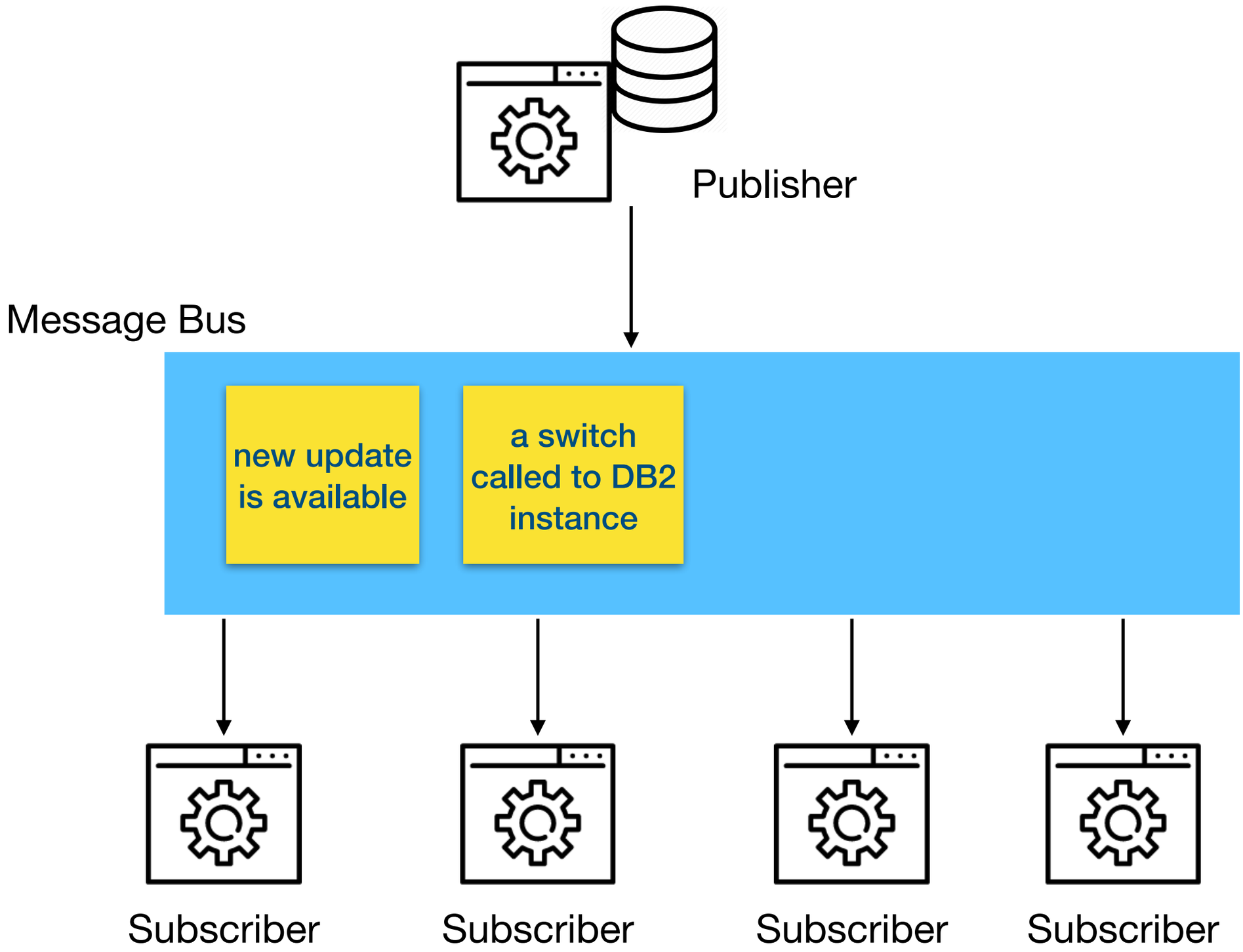
Messaging

- Messaging is a programmable functionality with enable both **asynchronous** and **synchronous** communication between different software components.

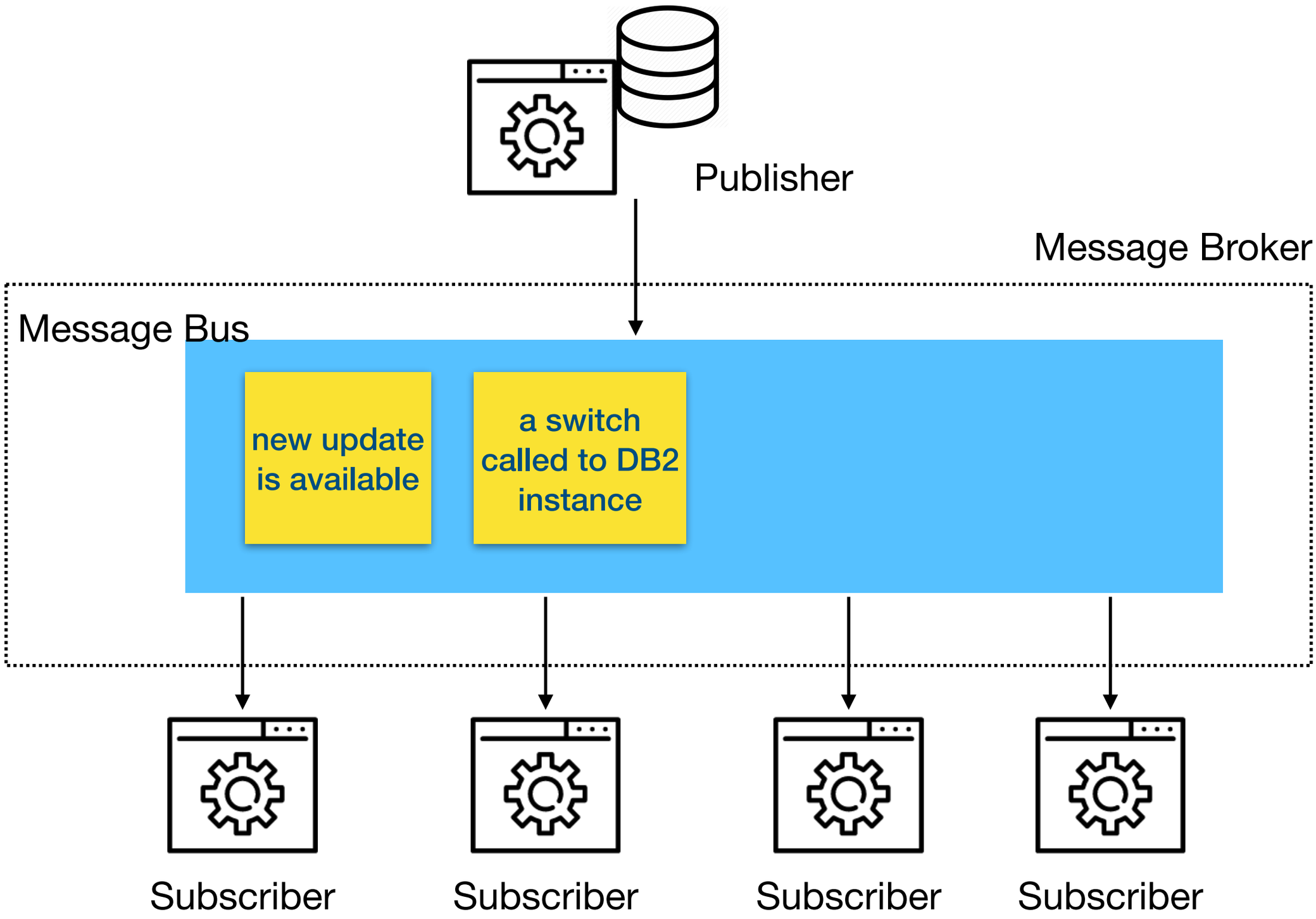
Outline

- Batch Processing
- Stream Processing
  - **Asynchronous Messaging**
  - Synchronous Messaging

Messaging


Publisher
Message Bus
new update is available
a switch called to DB2 instance
Subscriber
Subscriber
Subscriber
Subscriber

Messaging


Publisher
Message Broker
Message Bus
new update is available
a switch called to DB2 instance
Subscriber
Subscriber
Subscriber
Subscriber

Why Asynchronous Messaging?

Fast
Producer

Slow
Consumer1

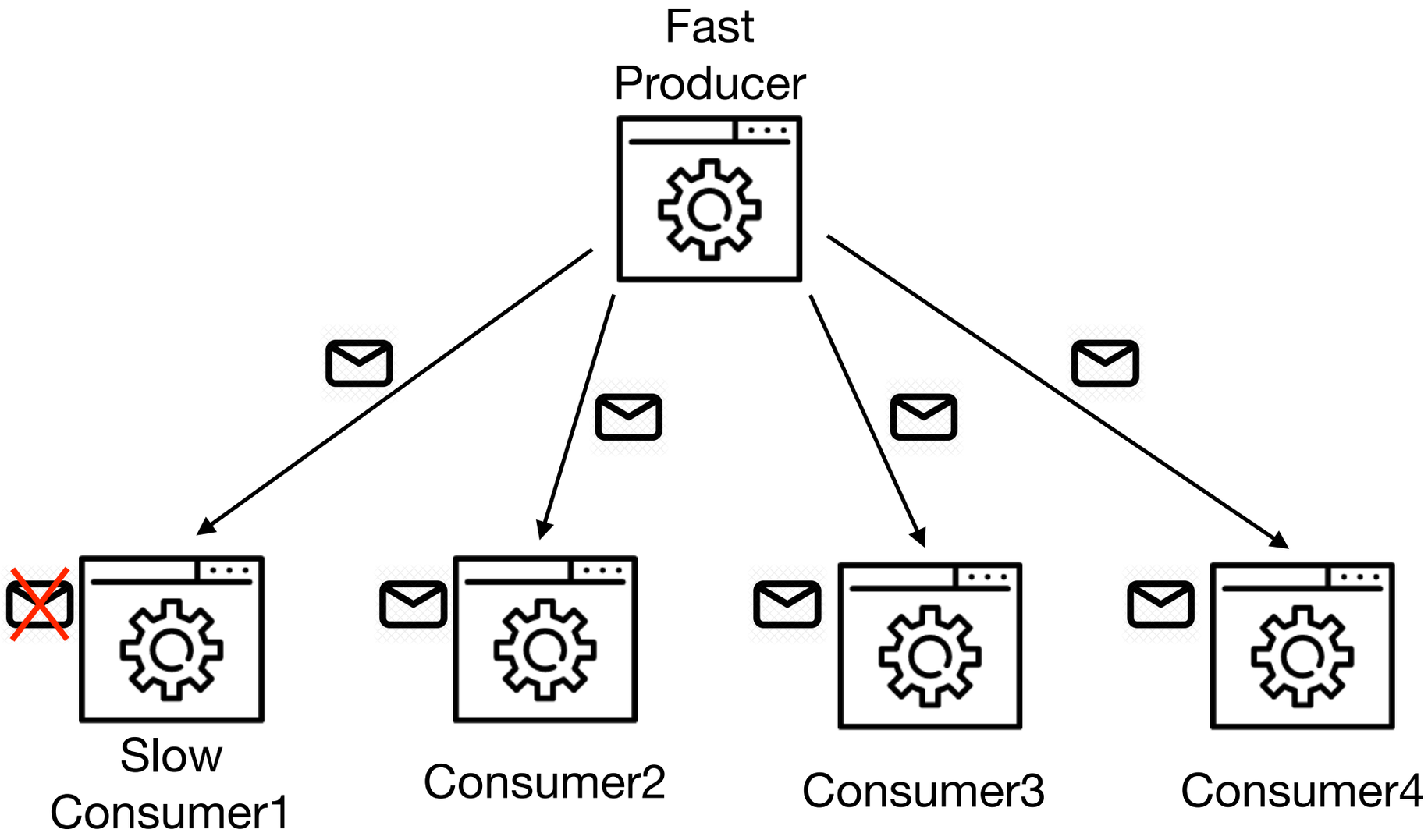
Consumer2

Consumer3

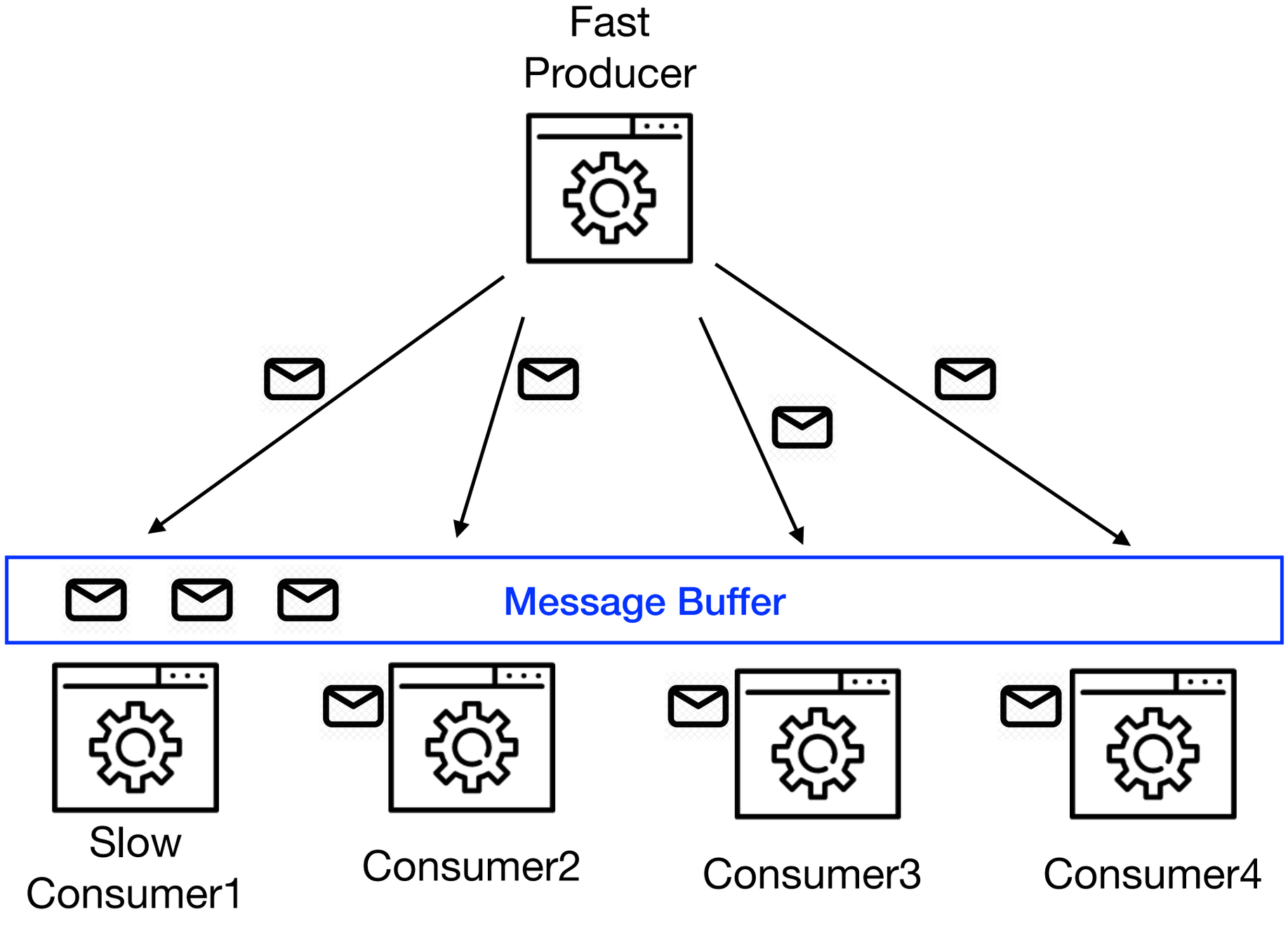
Consumer4

What happen when a receiver can not keep up with the sender?

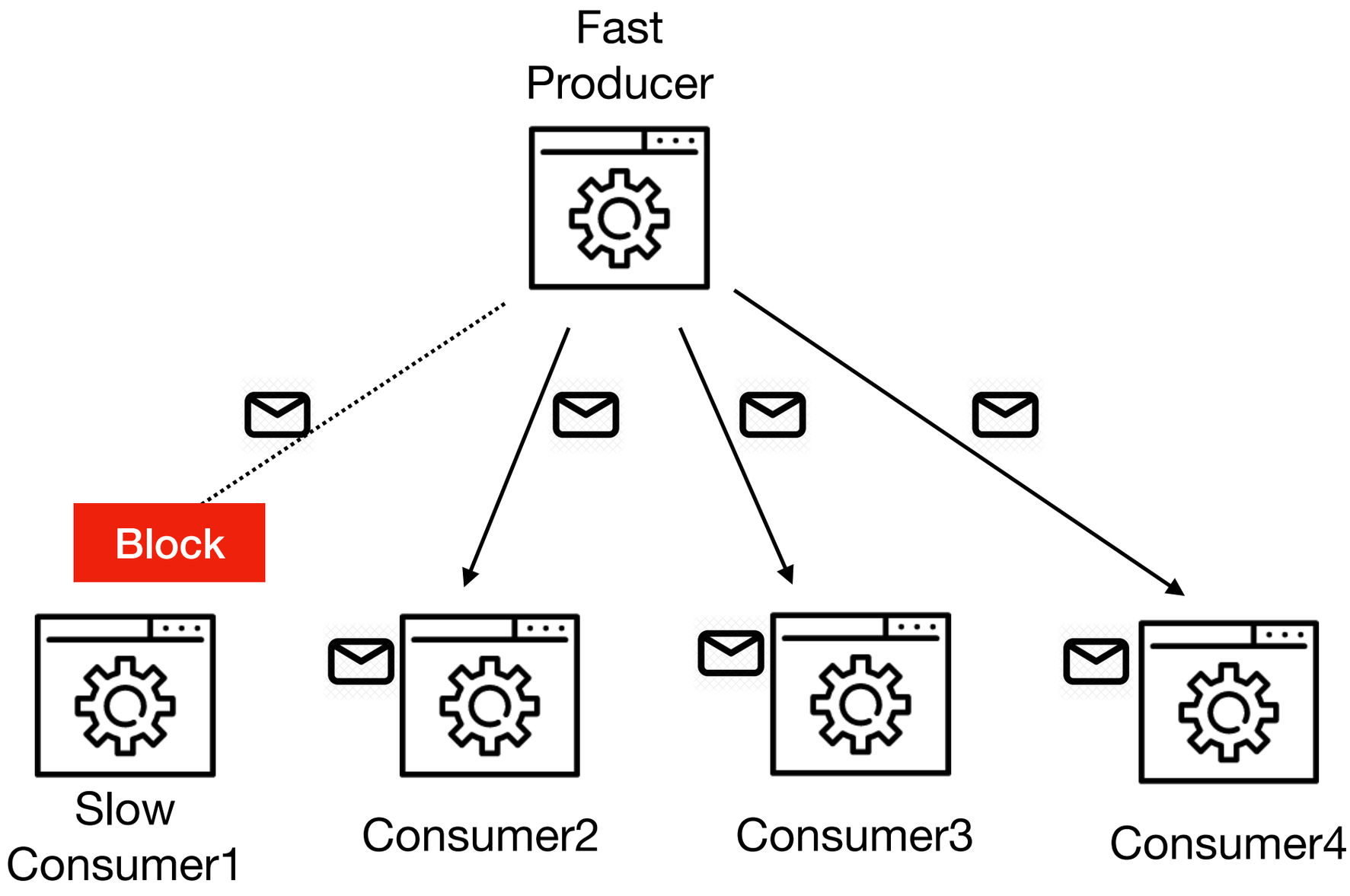
Scenario 1: Drop Message



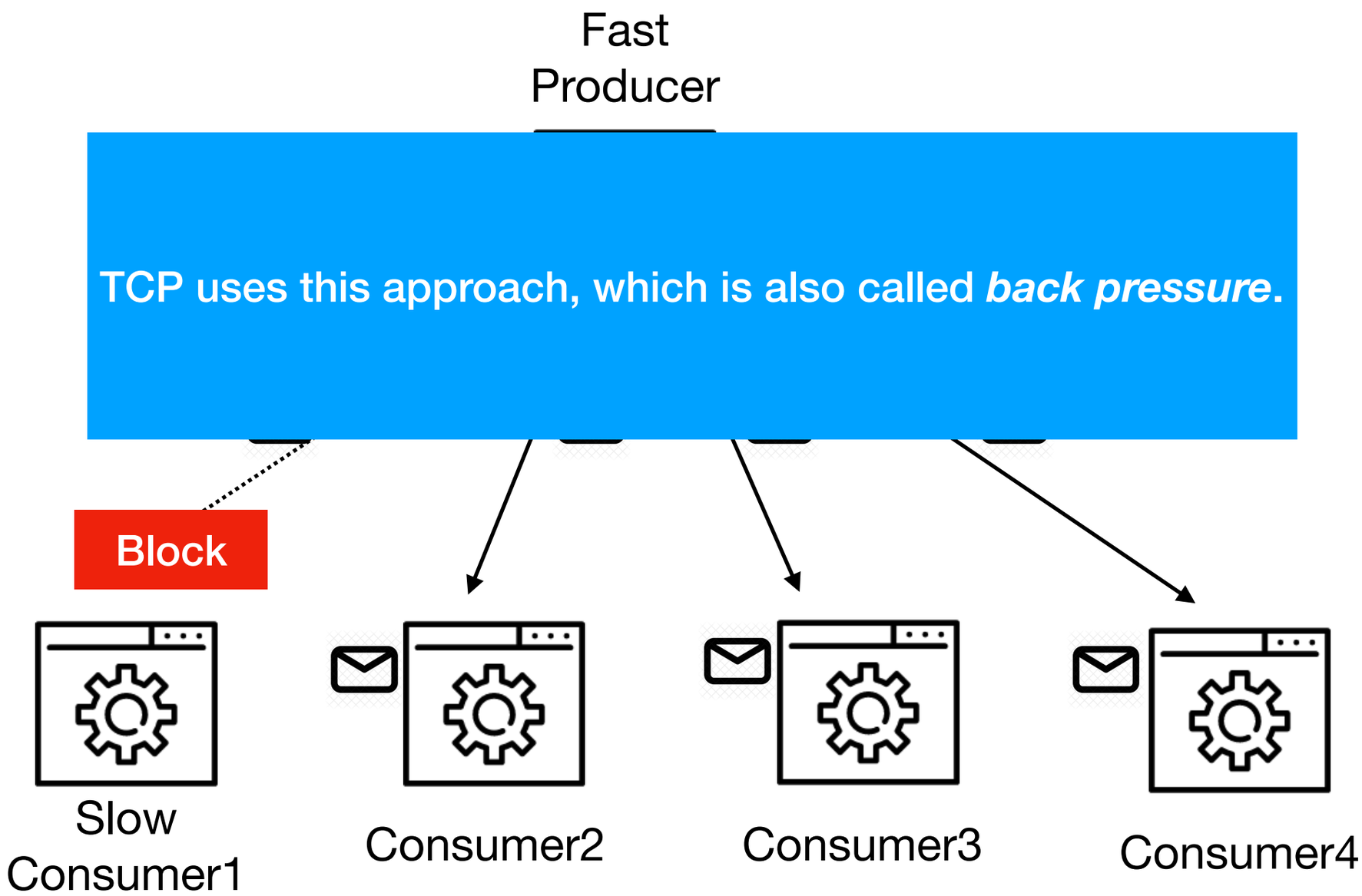
Scenario 2: Buffer Messages in a Queue



Scenario 3 Flow Control



Scenario 3 Flow Control



What happen if a system loses a message?

- To avoid loosing messages, we can keep a backup of messages on the disk, or use an auxiliary memory buffer (message queue).

- If our application can cope with loosing messages, usually it has a high throughput and low latency (fast response time)

- **Response time:** It is the time between requesting for something and receiving back the response.

- **Throughput:** The number of process that are completed per unit of time, is called throughput.

High Reliability in Batch Processing

- Batch processing does not have the challenge of losing a message.

- Failed tasks will be recalled and re-executed, because response time is not an issue. Somehow it shows easy fault tolerance.

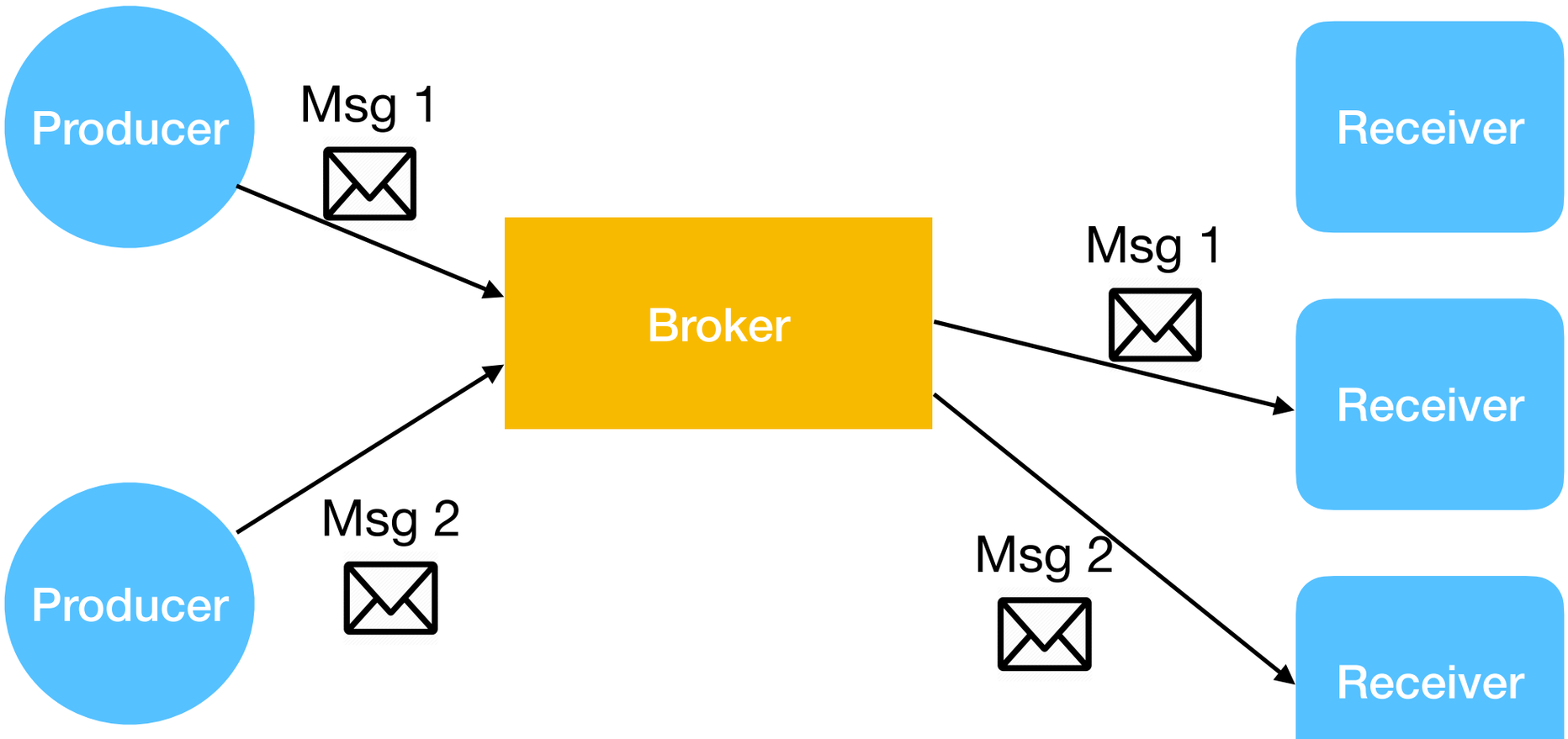
- Usually, batch jobs are usually **idempotent** operations. This means that multiple executions of a job has the same effect as we performed it once, e.g. sorting static data, storing a key-value data in a set that does not allow duplicate with the same key (value will be overwritten)

Multiple Receiver Patterns

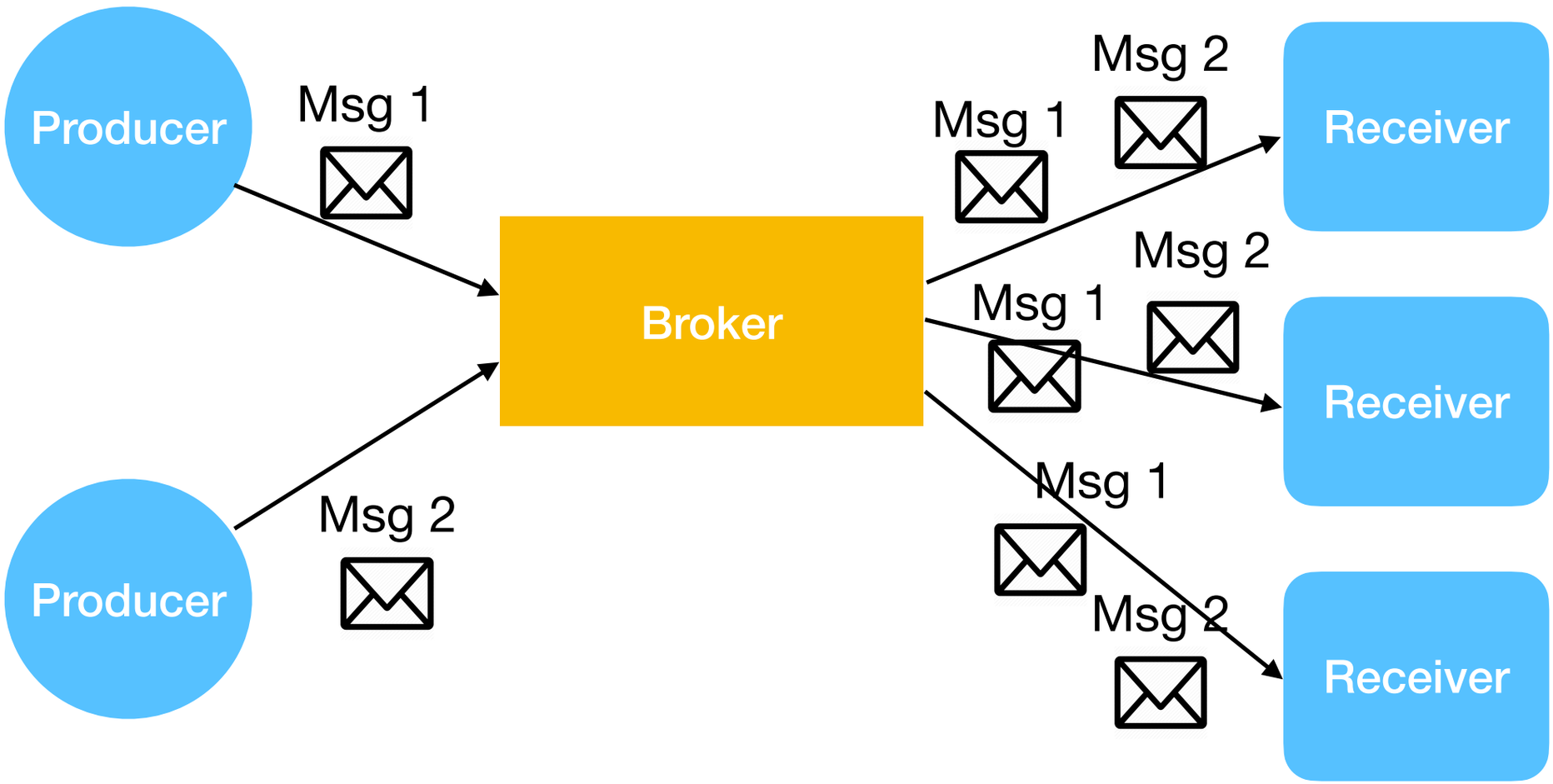
- **Load balancing:** When **processing a message is expensive** we can add more receiver nodes that enable them to process messages in parallel. This pattern of message consumption is called load balancing.

- **Fan-out:** A message is broadcasted and several receivers can receive that message, without affecting each other.

Load Balancing



Fan-out


Producer
Msg 1
Producer
Msg 2
Broker
Msg 1
Msg 2
Receiver
Msg 1
Msg 2
Receiver
Msg 1
Msg 2
Receiver

Message Acknowledgement

- The receiver may crash at any time, and thus the message receive process could be unsuccessful.

- Therefore, we need a way to notify the message broker whether the message has been delivered successfully.

- The receiver will send an **acknowledgment** upon successful message received to the broker. Then the broker realizes that it can remove the message from its message bus.

- Nevertheless, note some messaging protocols such as AMPQ **preserving require the order of messages**. To resolve this issue, we can use a separate message queue for each consumer.

Sending continuously acknowledgement even for small messages is resource consuming, what is your recommendation to mitigate this challenge?

S sly
acknow or small
messag suming,
what is ation to
mi ge?

1. Periodically sending back the acknowledgements and buffer the message acknowledgments on the receiver.

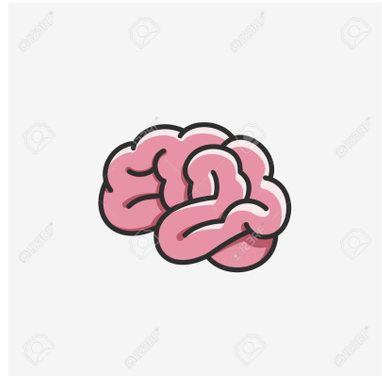
2. all messages might not required acknowledgement. Therefore, there is no need to send for every message an acknowledgment

Fault Tolerance in Stream Processing

- Fault tolerance in stream processing is not as easy as batch jobs.

- We can check point ID or use a time to label stream jobs and their order. When a job failed, the system should revert back to its predefined check point.

- Reverting back to checkpoints are also implemented in micro batching which is a sort of steam processing (despite the name might be confusing)

Keep this here: 

- *As with batch processing, we need to discard the partial output of any failed tasks. However, since a stream process is long-running and produces output continuously, we can't simply discard all output. Instead, a finer-grained recovery mechanism can be used, based on microbatching, checkpointing, transactions, or idempotent writes. [Klepman '18]*

Java Email Service

Sending Email with Java

1. Download `mail-1.4.7.jar` and add it into project libraries.

2. Configure the email service credential.

3. Create an email session (requires our email user name and password)

4. Create the “Multipurpose Internet Mail Extensions” (MIME) message

5. Send the message

Sending Email with Java

1. Download `mail-1.4.7.jar` and add it into project libraries.

2. Configu

3. Create name and pass

4. Create the "Multipurpose Internet Mail Extensions" (MIME) message

5. Send the message

Download and Run FakeSMTP to enable having SMTP mail client

Sending Simple Email with SMTP protocol

```
import javax.mail.*;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeBodyPart;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeMultipart;
import java.io.File;
import java.util.Properties;

public class SendingEmail {

    private String host = "";
    private int port = 0;
    private String username = "";
    private String password = “";

    private void sendMail(String host, int port, String username, String password) {
        Properties prop = new Properties();
        prop.put("mail.smtp.auth", true);
        prop.put("mail.smtp.starttls.enable", "true");
        prop.put("mail.smtp.host", host);
        prop.put("mail.smtp.port", port);
        prop.put("mail.smtp.ssl.trust", host);
        Session session = Session.getInstance(prop, new Authenticator() {
            @Override
            protected PasswordAuthentication getPasswordAuthentication() {
               return new PasswordAuthentication(username, password);
            }
        });
        try {
                Message message = new MimeMessage(session);
                message.setFrom(new InternetAddress("testsender@noemail.com"));
                message.setRecipients(Message.RecipientType.TO, InternetAddress.parse("rezar@bu.edu"));
                message.setSubject("Mail Subject");

                String msg = "This is a test email using JavaMailer";

                MimeBodyPart mimeBodyPart = new MimeBodyPart();
                mimeBodyPart.setContent(msg, "text/html");

                Multipart multipart = new MimeMultipart();
                multipart.addBodyPart(mimeBodyPart);

                message.setContent(multipart);
                Transport.send(message);

        } catch (Exception e) {
            e.printStackTrace(); }
    }

    public static void main(String ... args) {
        SendingEmail semail = new SendingEmail();
        System.out.println("start initiating the connection");
        semail.sendMail("localhost", 25, "someusername", "somepass");
        System.out.println("email send process is finished");
    }
} }
```

Sending HTML Email with SMTP protocol

```
import javax.mail.*;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeBodyPart;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeMultipart;
import java.io.File;
import java.util.Properties;

public class SendingHTMLemail {
      private void sendMail(String host, int port, String username, String password) {

            Properties prop = new Properties();
            prop.put("mail.smtp.auth", true);
            prop.put("mail.smtp.starttls.enable", "true");
            prop.put("mail.smtp.host", host);
            prop.put("mail.smtp.port", port);
            prop.put("mail.smtp.ssl.trust", host);

            Session session = Session.getInstance(prop, new Authenticator() {
                @Override
                protected PasswordAuthentication getPasswordAuthentication() {
                   return new PasswordAuthentication(username, password);
                }
            });
            try {
                    Message message = new MimeMessage(session);
                    message.setFrom(new InternetAddress("testsender@noemail.com"));
                    message.setRecipients(Message.RecipientType.TO, InternetAddress.parse("targetemail@nowhere.edu"));
                    message.setSubject("Mail Subject");
                    String msg = new String("<h1>This is the message in html format</h1> </br> </br> Goodbye");

                    MimeBodyPart mimeBodyPart = new MimeBodyPart();
                    mimeBodyPart.setContent(msg, "text/html");

                    Multipart multipart = new MimeMultipart();
                    multipart.addBodyPart(mimeBodyPart);

                    message.setContent(multipart);

                    Transport.send(message);

                } catch (Exception e) {
                    e.printStackTrace();
                }
            }

        public static void main(String ... args) {
                SendingHTMLemail semail = new SendingHTMLemail();
                System.out.println("start initiating the connection");
                semail.sendMail("localhost", 25, "someusername", "somepass");
                System.out.println("email send process is finished");
            }
}
```

Sending Email with Attachment

```
import javax.mail.internet.MimeBodyPart;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeMultipart;
import java.io.File;
import java.util.Properties;

public class SendingEmailAttachment {

    private String host = "";
    private int port = 0;
    private String username = "";
    private String password = "";

    private void sendMail(String host, int port, String username, String password) {
        Properties prop = new Properties();
        prop.put("mail.smtp.auth", true);
        prop.put("mail.smtp.starttls.enable", “true");
        prop.put("mail.smtp.host", host);
        prop.put("mail.smtp.port", port);
        prop.put("mail.smtp.ssl.trust", host);
        Session session = Session.getInstance(prop, new Authenticator() {
    @Override
    protected PasswordAuthentication getPasswordAuthentication() {
       return new PasswordAuthentication(username, password);
    }
});
try {
        Message message = new MimeMessage(session);
        message.setFrom(new InternetAddress("sender@attachment.com"));
        message.setRecipients(Message.RecipientType.TO, InternetAddress.parse("xxx@attachment.edu"));
        message.setSubject("I don't forget to include METCS622 in my email title");

     // Create a multipart message
        Multipart multipart = new MimeMultipart();

        BodyPart messageBodyPart = new MimeBodyPart();
        messageBodyPart.setText("This is sample text");

        // Set text message part
        multipart.addBodyPart(messageBodyPart);

        // Part two is attachment
        messageBodyPart = new MimeBodyPart();
```

Sending Email with Attachment

```
        @Override
        protected PasswordAuthentication getPasswordAuthentication() {
           return new PasswordAuthentication(username, password);
        }
    });
    try {
            Message message = new MimeMessage(session);
            message.setFrom(new InternetAddress("sender@attachment.com"));
            message.setRecipients(Message.RecipientType.TO, InternetAddress.parse("xxx@attachment.edu"));
            message.setSubject("I don't forget to include METCS622 in my email title");

         // Create a multipart message
            Multipart multipart = new MimeMultipart();

            BodyPart messageBodyPart = new MimeBodyPart();
            messageBodyPart.setText("This is sample text");

            // Set text message part
            multipart.addBodyPart(messageBodyPart);

            // Part two is attachment
                    messageBodyPart = new MimeBodyPart();
                    String filename = "/Users/rawassizadeh/WORK/eclipseworkspace/CSMET622/Introduction.pdf";
                    DataSource source = new FileDataSource(filename);
                    System.out.println("---->"+source.getContentType());
                    messageBodyPart.setDataHandler(new DataHandler(source));
                    messageBodyPart.setFileName(filename);
                    multipart.addBodyPart(messageBodyPart);
                    message.setContent(multipart);
                    Transport.send(message);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        public static void main(String ... args) {
            SendingEmailAttachment semail = new SendingEmailAttachment();
            System.out.println("start initiating the connection");
            semail.sendMail("localhost", 25, "someusername", "somepass");
            System.out.println("email send process is finished");
        }
    }
```

Outline

- Batch Processing
- Stream Processing
  - Asynchronous Messaging
  - **Synchronous Messaging**

Synchronous Messaging

- It is a style of communication between two software component that one component is waiting for another component to respond.

- Email and web messages are two good example of asynchronous messaging.

Java Supports for Synchronous Messaging

- Packet based messaging (Use UDP for data transfer)
- Stream based messaging (Use TCP for data transfer)
- TCP: Transfer Control Protocol
- UDP: User Datagram Protocol

Java Supports for Synchronous Messaging

- TCP connection is session aware, but UDP is session-less.

- There is **no guarantee that UDP packets arrive correctly**, also there is **no guarantee on the delivery of the UDP packet**.

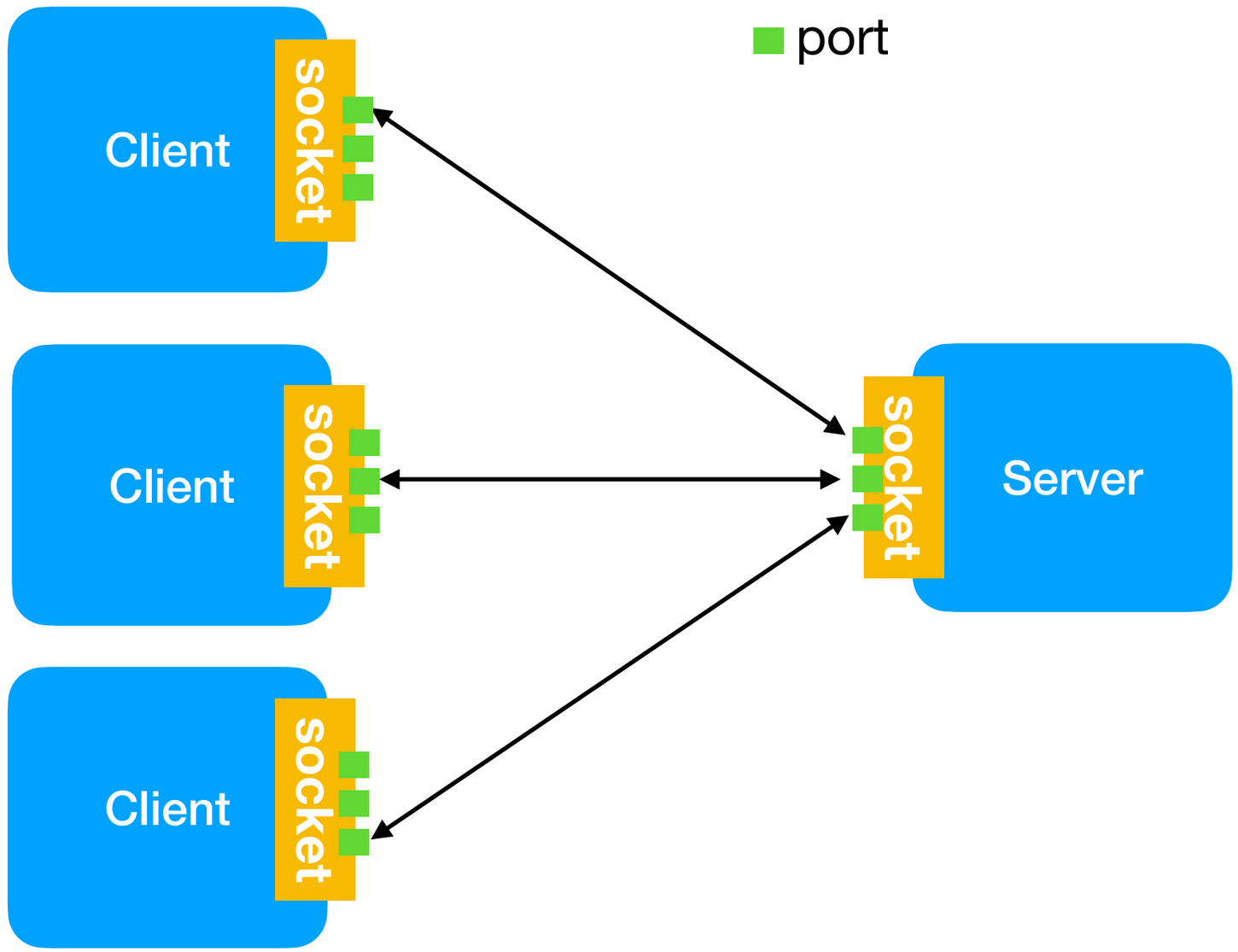
- Nevertheless, **UDP is much faster than TCP**, because there is no overhead. It is useful for communications that don’t require a guarantee for message delivery.

- For example, assume we would like to distribute a **video**. In this scenario, few frame delays can be tolerated and there is no need to guarantee all message delivery. Therefore, UDP is favored over TCP.

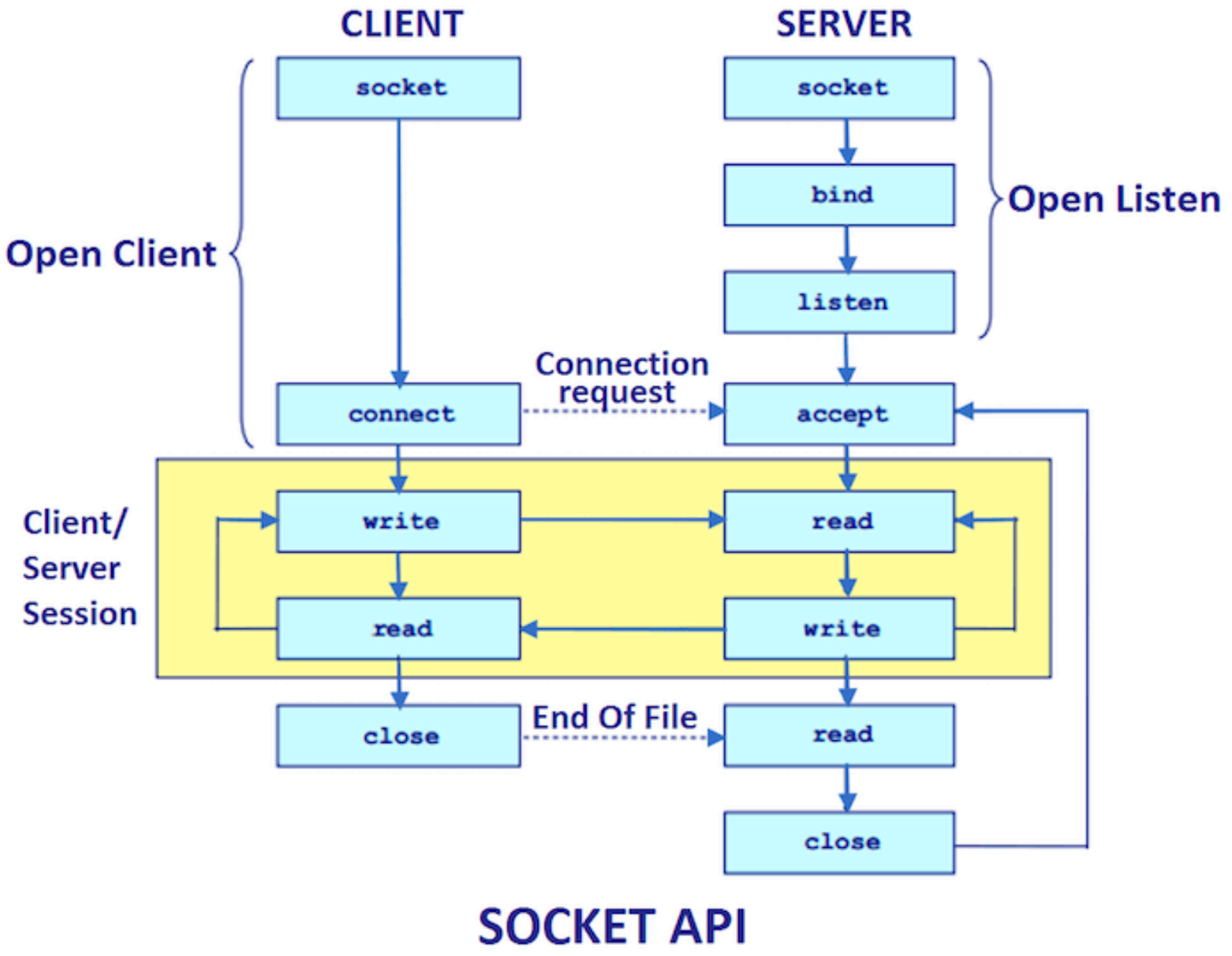
Socket Based Communication

- Client and server program could be located on a separate machines, but they could be also running on the same computer.

- In our scenario your client and server codes both reside on the same machine (your machine).



Socket Based Communication Workflow



Source: https://www.javatpoint.com/socket-programming

Server:

- The first step is to create a server socket.

```
ServerSocket serversocket = new ServerSocket(9000);
```

- After the successful socket creation, the server should start to wait for a connection.

```
Socket socket = serversocket.accept();
```

Client:

- The client should request a server for a connection with server name and port.

```
Socket socket = new Socket(servername, port);
```

HelloWorld Server Socket

```
package edu.bu.met622.socket;

import java.io.DataInputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class SimpleServer {
   public static void main(String[] args) {
      try {
         ServerSocket serversocket = new ServerSocket(9000);
         System.out.println("[Server] Server started ... waiting ...");
         Socket socket = serversocket.accept();
         DataInputStream dis=new DataInputStream(socket.getInputStream());
         String str=(String)dis.readUTF();
         System.out.println("[Server] the message that the server recieved
is:"+str);
         socket.close();
      }catch(IOException ex) {
         ex.printStackTrace();
         System.out.println("[Server] ERROR:"+ex.getMessage());
      }
   }
}
```

HelloWorld Client Socket

```
package edu.bu.met622.socket;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class SimpleClient {
   public static void main(String[] args) {
      try {
         Socket s = new Socket("localhost", 9000);
         DataOutputStream dout = new DataOutputStream(s.getOutputStream());
         dout.writeUTF("Hello 4m Client");
         dout.close();
         dout.flush();
         s.close();
      } catch (IOException ex) {
         System.out.println("[Client] Error:" + ex.getMessage());
         ex.printStackTrace();
      }
   }
}
```

HelloWorld Client Socket

```
package edu.bu.met622.socket;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class SimpleClient {
    public static
        try {
            Socket s
            DataOutpu                                    tOutputStream());
            dout.wri
            dout.clo
            dout.flu
            s.close(
        } catch (IOException ex) {
            System.out.println("[Client] Error:" + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

Be careful with the port number, it is arbitrary but always check the default ports before you open one

Another Example

In this example, first, the client will write to the server. Then, the server receives and prints the text. Next, the server will write to the client and the client will receive and print the text, and it goes on, until the server receives “stop”.

Server

```java
package edu.bu.met622.socket;

import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;

public class AnotherServer {
    public static void main(String args[]) throws Exception {
        ServerSocket ss = new ServerSocket(9000);
        Socket s = ss.accept();
        DataInputStream din = new DataInputStream(s.getInputStream());
        DataOutputStream dout = new DataOutputStream(s.getOutputStream());
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str = "", str2 = "";
        while (!str.equals("stop")) {
            str = din.readUTF();
            System.out.println("client says: " + str);
            str2 = br.readLine();
            dout.writeUTF(str2);
            dout.flush();
        }
        din.close();
        s.close();
        ss.close();
    }
}
```

Client

```java
package edu.bu.met622.socket;

import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.InputStreamReader;
import java.net.Socket;

public class AnotherClient {
    public static void main(String args[]) throws Exception {
        Socket s = new Socket("localhost", 9000);
        DataInputStream din = new DataInputStream(s.getInputStream());
        DataOutputStream dout = new DataOutputStream(s.getOutputStream());
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str = "", str2 = "";
        while (!str.equals("stop")) {
            str = br.readLine();
            dout.writeUTF(str);
            dout.flush();
            str2 = din.readUTF();
            System.out.println("Server says: " + str2);
        }
        dout.close();
        s.close();
    }
}
```

A Multi-Thread Server which can Handle Several Clients

```
package edu.bu.met622.socket;

import java.io.*;
import java.net.*;
import java.util.*;

public class MultiThreadServer {
    public static void main(String[] args)
    {
        try {
            ServerSocket serversocket = new ServerSocket(9000);
            System.out.println("Server says: Server started ... waiting ...");

            while (true)
            {
                Socket socket = serversocket.accept();
                InetAddress inetad = socket.getInetAddress();
                System.out.println("Client IP address: " + inetad.getHostAddress() +
                        " host name: " + inetad.getHostName());
                new Thread(new ProcessingSingleClient(socket)).start();
            }
        } catch (IOException excep) {
            excep.printStackTrace();
        }
    }
}
```

Server 2nd Class to Process Client

```
package edu.bu.met622.socket;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
/**
 * This codes belongs to server and it is not a client
 */
public class ProcessingSingleClient  implements Runnable
{
    private Socket socket;
    public ProcessingSingleClient(Socket s) {
        socket = s;
    }
    public void run()
    {
        try {
            System.out.println("Thread Id = " + Thread.currentThread().getId() + " start");
            DataInputStream inputfromclient = new DataInputStream(socket.getInputStream());
            DataOutputStream outputtoclient = new DataOutputStream(socket.getOutputStream());
            int j = inputfromclient.readInt();
            System.out.printf("Client: client sent %d%n", j);
            outputtoclient.writeInt(j * j);
            System.out.printf("Client: client got back %d%n", j * j);
            System.out.println("Thread Id = " + Thread.currentThread().getId() + " end");
        }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

Client

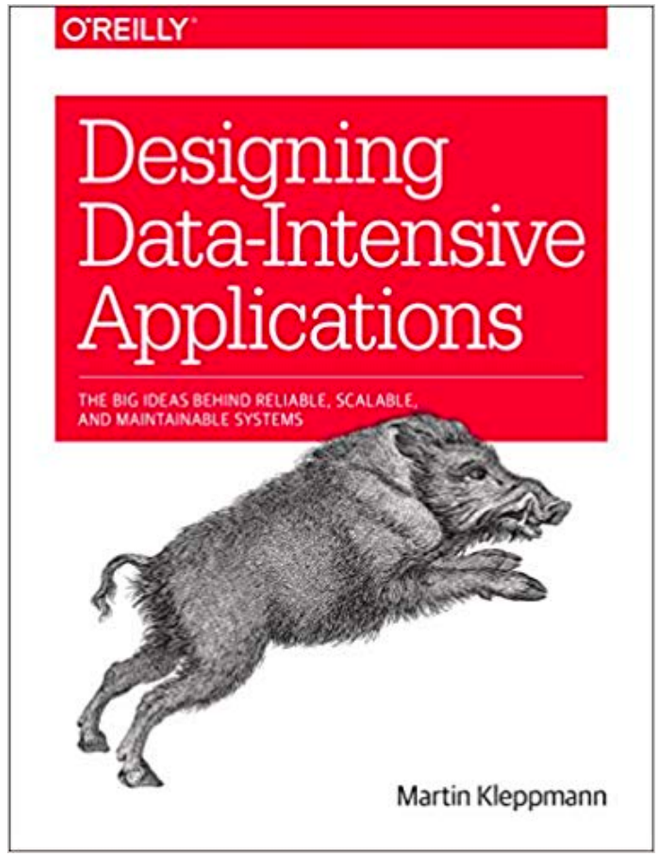
- Just use one of the old client method, e.g. SimpleClient.

References

https://dzone.com/articles/which-are-the-iot-messaging-protocols

https://www.rabbitmq.com/tutorials/tutorial-one-java.html

https://www.baeldung.com/a-guide-to-java-sockets




BIG FAT
THANK YOU