TSR NOTES

UNIT 1: INTRODUCTION

1.-CONCEPT OF DISTRIBUTED SYSTEM

A **distributed system** is a set of autonomous agents, in which each agent is a sequential process, proceeding at its own pace. These agents interact using:

- **Message passing**.
- **Shared memory**.

Agents have their own independent state. There is some collective goal to this cooperation. In practice, a **distributed system** is a **networked system**.

2.-RELEVANCE

The main **reasons** of its **relevance** are (from the 80's):

- **Speed up**.
- **Fault tolerance**.
- **Resource sharing**.

All the previous reasons are still valid today because the **computing environment** is **distributed** and **interconnected**:

- Myriad of connected "computers".
- Myriad of remote services.

The **challenges** are:

- Leverage the connectivity to achieve useful results.
- Create subsystems capable of delivering well-behaved services.

3.-APPLICATION AREAS

Some important application areas are:

- **WorldWideWeb**.
  - It allows access to remote documents.
  - It allows to request remote services.
- **Sensor networks**. Collect data from the environment.
- **Internet of Things**. Sensors and actuators interact with the environment in order to have task automatization.
- **Cooperative computing**. Parallel execution of a large computing task.
- **Highly available Clusters**. Ensure service continuity.
- **Cloud Computing**.
  - SaaS.
  - IaaS.

4.-BACK TO THE BEGINNING: CLOUD COMPUTING

4.1.-PROGRAMS AND SERVICES

The **main goal** of **CC** is to make creation and exploitation of services based on software simpler and more efficient.

A **program** is a static entity generated by development.

A **service** is when a program or application is running, the functionality being provided by the corresponding processes.

4.2.-ROLE IN THE SERVICE CYCLE

The main four roles are:

- **Developer**: the implementor of the application.
- **Service provider**: the person or company that decides which programs are needed for providing a service. It decides the characteristics of a service, the components that make it up, and how it should be configured and managed.
- **System's administrator**: manages software deployment and configuration. Software upgrades shouldn't compromise service continuity.
- **Service user**: the consumer of a service.

4.3.-EVOLUTION OF SOFTWARE SERVICES

4.3.1.-MAINFRAMES

Main features:

- System administration is **taken care** of by **specialists**.
- Very **few tensions of contention**.
- **Efficient** use of **hardware**.
- **Mixed role for users**:
    - Administrators.
    - Developers.
    - Service providers.
    - Users.
- **Users** were **involved** in **too many** of the management details for the services they finally used.
- The **perfect user** is the **head of department**.

4.3.2-PERSONAL COMPUTERS/WORKSTATIONS

Main features:

- Trends with **increased computer power** produced the personal computer.
- **No contention**.
- **Wasteful** use of **resources**: computer infra-utilized.
- Up-front **investment cost** to purchase.
- **Rationalization of the role of developer**. Specialized organizations build the software.
- **Mixed role for users**.
- **Too much complexity** for the majority of users.

4.3.3.-ENTERPRISE COMPUTER CENTERS

Main features:

- Similar characteristics to the PC situation.
  - **Expensive**.
  - **Infrautilization**.
- **Ask for the services** to specialised companies.

4.3.4.-SOFTWARE AS A SERVICE (SAAS)

Main features:

- **Service** is **accessed** through the **network**.
- **Clear separation** of the role of **user**.
- **Not** so clear **separation** of the **other roles**.
- Initially some inefficiencies:
  - **Lack of flexibility** in hardware allocation.
  - **Limited contention**.

4.3.5.-INFRAESTRUCTURE AS A SERVICE (IAAS)

Main features:

- Provides ability to **easily allocate/de-allocate** computer and network **resources** on demand.
- Driven by **Hardware Virtualization technology**.

4.3.6.-SAAS ON IAAS

Main features:

- Makes it **feasible** to **create SaaS** which adapt to their user's load.
- Very **efficient usage of resources** for SaaS providers.
- **IaaS** providers take the **risk** of up-front **investment**.
- **SaaS** provider still has **mixed roles**.

4.3.7.-PLATFORM AS A SERVICE (PAAS)

Main features:

- Ideally promises to **take away extraneous tasks** from SaaS providers.
- Purports to be the **equivalent** of an **OS**.
- Software deployments **automatization**:
    - **Installation**.
    - **Configuration**.
    - **Upgrading**.
    - **Management of variables workloads**.

4.3.8.-SUMMARY

Cloud Computing is all about efficiency and easiness.

Three layers of cloud services are identified:

- SAAS.
- PAAS.
- IAAS.

5.-PROGRAMMING PARADIGMS

To be able to scale, servers must not BLOCK while serving requests.

Two **programming paradigms** exist:

- **Multi-threaded programming**: state-sharing concurrent servers.
- **Asynchronous programming**: asynchronous single-threaded servers.

5.1.-CONCURRENT, STATE-SHARING SERVERS

In **multi-threaded** concurrent programs:

- Each request is handled in its own thread.
- All threads share a global state.
- Concurrency-control mechanisms are used to implement atomicity.

The main **advantage** is that threads may block waiting for requests from the server to complete, **without blocking the server**.

The **disadvantages** are:

- Multi-threaded programming has its own **overheads**. It needs to support concurrency control constructs.
- It turns out that shared memory concurrent programming is **hard** to do well and to reason about how it works.
- **Prevalent Environments**.

5.2.-ASYNC SERVERS

In **async programming** aka event-driven programming:

- Closely matches the guard-action program model.
- Async programs have many activities, but state is never concurrently shared among the executing activities.

"**Events**" are the "**guards**".

Actions are established as callbacks of events. They need to dynamically built actions/guards to facilitate programming. When building actions, programming language mechanisms are used to easily establish the state to be affected by the action, that it reduces complexity of "preparing" state to link internal actions.

Actions ready for execution are placed on a "**turn queue**" and scheduled actions are executed in **FIFO** order of the queue.

The **advantages** are:

- **Shared state handling complexity** greatly **diminishes**. Still, careful on handling of the turn queue to avoid surprises.
- **Less overhead**, as no multi-threading environment is supported, so it has a better ability to scale.
- Close match to how a distributed system actually works: **event-driven**. It is easier to reason about what is going on.

The main **disadvantages** are:

- **Proper state-hand**ling is necessary when building actions.
- Needs all **environment** to be **async**, not just IPC. OS services need to be async too, to avoid blocking.

6.-CONCLUSIONS

- Networked Systems are Distributed Systems
    - Most computing nowadays is networked
    - Proper design and implementation requires mastering aspects of concurrent programming and properties of the architecture
- Rich set of application areas already exploited
- Important trend of Cloud Computing as a logical endpoint in the evolution of computing
    - Driven by efficiency in resource usage
    - Pay-as-you-go model of access
    - Elasticity and scalability
- Two programming paradigms for distributed service development:
    - Concurrent (i.e., multi-threaded) servers. Should deal with race conditions. Threads may be blocked.
    - Asynchronous servers. Event-driven. Easily scalable.

UNIT 2: JAVASCRIPT AND NODEJS

1.-INTRODUCTION

JavaScript is a scripting language, interpreted, dynamic and portable. It has a high level of abstraction, thus:

- **Simple programs**.
- **Fast development**.

This programming language was initially designed for providing dynamic behaviour to web pages. Current browsers include an interpreter of this language. It is **event-driven** with **asynchronous interactions** supported with “callbacks”, this makes a boost both in throughput and scalability. It **hasn’t support** for **multi-threading**, so:

- **No shared objects**. No need for synchronisation mechanisms.
- We should take care about when a variable gets its value, that is, **callback management**.

It **supports** both **functional** and **object-oriented programming**.

NodeJS is a development platform based on the JavaScript interpreter being used by Google in its Chrome browser. NodeJS provides a series of modules that facilitates the development of distributed applications.

2.-JAVASCRIPT

JavaScript main characteristics are:

- **Imperative** and **structured**, so the syntax is similar to that of Java.
- Multi-paradigm:
  - **Functional programming**: functions are “objects” and can be used as arguments to other functions.
  - **Object-oriented programming**: based on prototypes, instead of regular classes and inheritance. However, prototypes may emulate object orientation.
- Related to other programming languages.

There are two basic alternatives to run programs:

1. Using the interpreter included in web browsers.
2. Using an external interpreter.

JavaScript is **not** a “**strongly typed**” language. A variable is declared (with “let” or “var”) before its first use, but without any specification of type. A variable may hold, in an execution, elements of different types. Objects are heterogeneous, so their values may belong to different types.

JavaScript **type management** is **weak**, so we should take care of its implicit type conversions.

The **scope of a variable** is:

- **Local to the block** where it has been declared (using let)
- **Local to the function** where it has been declared (using var)
- **Global** (entire file) when:
    - It is not declared inside a function. Equivalent to assume an implicit global function that holds the entire file.
    - It is declared in a function without using let or var.

A statement may access all variables that have been defined in the scopes that include that statement. Variables are searched from the inner to the outer scope.

There are two **types of functions**:

- **Anonymous functions**: function(args){…}. They are a value that can be assigned, passed as an argument, etc. They are invoked as identifier(args), returning a single value.
- **Declaration functions**: function name(args){…}. Equivalent to: const name = function (args){…}.

Functions can be declared everywhere, even inside another function. They provide the scope for variable definition, when variables are defined using var. **Arguments** are **passed by value**, but **objects** are actually **passed by reference**. Functions are objects with properties and methods. They have a single return value, but it may be a composed element.

The **arity** is the number of arguments. A function with n arguments may be invoked using:

- Exactly n values.
- Less than n values. The remaining arguments receive the "undefined" value.
- More than n values. The unexpected arguments are ignored.

Arguments are accessed by name or using the "arguments" pseudo-array. The arity may be enforced, or default values may be assigned. There cannot be two functions with the same name, even when they are defined with different arities.

The **closure** equals the function plus the connection to variables in outer scopes. Functions remember the scope where they have been created.

A "**callback function**" is a reference to a function that is passed as an argument to another function B. B invokes that callback when it is terminating its execution. Callback functions allow asynchronous invocation. Callback nesting is not restricted. However, there are practical limits:

- Exceptions in nested callbacks. If an exception is not catched, it is propagated to the caller.
- If we do not guarantee a uniform management in all operations, some exceptions may be lost or managed in unexpected operations.
- The code is hard to read. The execution order is not intuitive.
- Uncertainty on the turn in which the callback will be run. We cannot rely on its execution in a concrete turn.

Asynchronous executions may be also built using promises. Operation calls follow the traditional format (easy to read) and there is no callback argument. The result of that call is a "**promise**" object. It represents a future value on which we may associate operations and manage errors. It may be in one of the following states:

- **Pending**: initial state. The operation has not yet concluded (unknown result).
- **Resolved**: the operation has terminated and we can get its result. This is a final state that cannot change.
    - **Rejected**: the operation has terminated with error. The reason is given.
    - **Fulfilled**: the operation has terminated successfully. A value is returned.

A function is associated to each final state (rejected vs fulfilled). Such function is run when the main thread finishes its current turn. Actually, it is enqueued in a new turn as a future event.

JavaScript is single-threaded, but multiple activities may be executed setting them as events. There is an event queue that accepts external interactions, holds pending activities and is turn-based. Each kind of event may be managed in a different way, but all event answers are executed by the same thread. This imposes a sequence-based management, that is that a new event isn't processed until the current one is finished.

3.-NODEJS

NodeJS is a special JavaScript interpreter:

- **Independent**. Valid for writing server agents.
- Most methods in Node.js modules allow **asynchronous interactions**:
    - Method returns immediately.
    - Results are provided via "callbacks".
    - An "asynchronous programming" model is followed:
        - Single-threaded: no concurrency, no shared variables, no critical sections. Very efficient. No concurrency "dangers". This single thread is not blocked in I/O operations nor when other traditionally blocking OS services are called.
    - Those asynchronous methods also have other blocking versions (without "callbacks").

Programmers see a single thread, but a queue of "function closures" is handled by the node runtime. It is the "turn queue". At each time, the Node runtime dequeues the first turn and executes it. This action defines a "turn". Note that setTimeout(f,0) stores function f() in the queue. It is useful when we need to execute f() once the current activity was finished.

Asynchronous modules are based on the libuv [7] library. libuv maintains a "thread pool".

When a blocking operation is called:

1. A thread T is taken from the "pool".
2. Invocation arguments are given to T, including the "callback" scope.
3. The invoking thread returns and our program goes on. T remains in the "ready-to-run" state.
4. T executes all operation sentences. It might block in some of them.
5. When T finishes that operation, it calls its associated "callback":
    a. T creates a scope for such "callback", passing the needed arguments.

b. T stores such scope in the turn queue.
c. T comes back to the "pool".
d. The "callback" is executed in a future turn. When it becomes the first in the turn queue. This avoids any race condition.

Exports: programmers should decide which objects and method are exported by a module. Each of those elements should be declared as a property of the "module.exports" object (or, simply, "exports").

Require: modules are imported using "require()". The module global object may be assigned to a variable. This names its context/scope.

The **events** module is needed for implementing event generators. Generators should be instances of EventEmitter. The event emitter should be created using "new". A generator throws events using its method emit(event [,arg1][,arg2][...]). emit() executes the event handlers in the current turn. If we do not want such behaviour, we should use: setTimeout(function() {emit(event,...);},0).

Event "listeners" may be registered in the event emitters using method **on(event,listener)** from the emitter. **addListener(event, listener)** does the same. The "listener" is a "callback" and is invoked each time the event is thrown. There may be multiple "listeners" for the same event.

Stream objects are needed to access data streams. There are four variants:

- **Readable**: read-only.
- **Writable**: write-only.
- **Duplex**: allow both read and write actions.
- **Transform**: similar to Duplex, but its writes usually depend on its reads.

All they are EventEmitter. Managed events:

- **Readable**: readable, data, end, close, error.
- **Writable**: drain, finish, pipe, unpipe.

The "net" module was made for the management of TCP sockets:

- **net.Server**: TCP server.
  - Generated using **net.createServer([options,][connectionListener])**. "connectionListener", when used, has a single parameter: a TCP socket already connected.
  - Events that may manage: listening, connection, close, error.
- **net.Socket**: Socket TCP.
  - Generated using "**new net.Socket()**" or "**net.connect(options [,listener])**" or "**net.connect(port [,host][,listener])**"
  - Implements a Duplex Stream.
  - Events that may manage: connect, data, end, timeout, drain, error, close.

The HTTP module was made to implement web servers (and also their clients). Consists of the following classes:

- **http.Server**: EventEmitter that models a web server.

- **http.ClientRequest**: HTTP request. It is a Writable Stream and an EventEmitter. Events: response, socket, connect, upgrade, continue.
- **http.ServerResponse**: HTTP response. It is a Writable Stream and an EventEmitter. Events: close.
- **http.IncomingMessage**: It implements the requests (for the web server) and the responses associated to ClientRequests. It is a Readable Stream. Events: close.

UNIT 3: MIDDLEWARE. ZEROMQ

1.-INTRODUCTION

Large Scale Distributed Systems Components are:

- **Specified** and **Developed independently**.
- **Deployed autonomously**. Each one designed as a Distributed Systems Agents, but establishing dependencies among themselves, consuming or providing functionality from/to other agents.
- **Even close-knit systems**. Developed by more than one developer. Multiple components clearly interdependent.

They need to easily interact usefully. In all cases:

- Need to deal with the complexities of the distributed environment.
- The product needs to be as error-free as possible.
- Submission to strict development schedule.

In order to make sure that we do not need absolute geniuses we have to write the millions of lines of code needed and to manage the systems deployed. The problem with this is the **complexity of the details**:

- Dealing with complexity is a source of errors.
- Dealing with too many details at once is a recipe for disaster.

Another problem is the fact that **many tasks are repetitive**, so we can save resources by providing **common implementations**.

The complexity sources come mainly from **making requests to a service**. Thus, communication is related to:

- **Finding servers implementing services**.
- **Specifying the functionality of services**.
    - Formatting information transmitted.
    - Synchronization between requester and server.
    - Security.
    - Failure handling.

There are two **complexity-taming approaches**:

- **Standards** (de facto and de jure):
    - Introduce streamlined ways to do thing.
    - Helps build familiarity for developers.
    - Help interoperability.
    - Providing High Level functionality.
- **Middleware**.

2.-MIDDLEWARE

The **middleware** is/are the layer(s) of software and services between application software and Network/Communications layers. It introduces various “transparencies”.

Transparency is the reduction of complexity by hiding and dealing with details in a uniform way.

Developer’s perspective:

- Easy to write.
- Reliable result.
- Maintainable.

Manager’s perspective:

- Easy set-up, change.
- Easy/possible to interact with products from third parties (interoperability).

3.-MESSAGING SYSTEMS

Messaging systems are **asynchronous** in nature, decoupling between sender and receiver. They send discrete pieces information with **arbitrary sizes** and that must be **transmitted atomically** (all or nothing). These pieces of information support for structuring messages and can be **queued** with some guarantee of order. Messaging systems don’t have a shared imposed view, so it is potentially better to produce scalable systems and to avoid concurrency difficulties.

There are two main approaches to messaging systems:

- **Transient** (stateless) systems: require the recipient to be up to receive a message.
- **Persistent**: messages are saved in buffers, thus, the recipient does not have to be up at sending time.

There are also two approaches within the **persistent** implementations:

Broker-based: specific servers store messages and provide strong guarantees. The is an overhead derived from the need to persistently store in secondary storage.

Broker-less: senders/receivers maintain the queues, typically in memory. Weaker persistence is guaranteed, but still, there is a decoupling between sender and receiver readiness to send/receive. It can be used to build broker-based system when needed.

4.-ZEROMQ

4.1.-INTRODUCTION

G**oals of ZeroMQ**:

- **Simple communications** middleware.
    - Easy configuration.
    - Easy & familiar to use.
- **Widely Available**. Portable implementation.
- **Support basic patterns**.
    - Eliminate need for each developer to re-invent the wheel.
    - Make it very useful right away.
- **Performance**.
    - No unnecessary overhead.
    - Tradeoff between reliability and performance.
- The **same code** can be used to **connect** (only URL changes needed):
    - Threads within a process.
    - Processes, within a machine.
    - Machines, through an IP network.

The main characteristics of ZeroMQ are:

- **Message-based**. Weakly persistent: main memory queues.
- It is just a **library**. No need to start specific middleware servers anywhere. Implemented in C++.

This technology provides sockets to send and receive messages (send/receive, bind/connect interface for a socket). It can use the following transports:

- TCP/IP.
- IPC.

The transport is used to instantiate sockets, so it can be easily changed by a configurations change.

Steps of a ZeroMQ process:

1. Application links with ØMQ library.
2. ØMQ maintains in-memory queues.
    a) At sender.
    b) At receiver.
3. ØMQ uses communication layers.

ZeroMQ must be installed before its usage in process since it is a library. In order to use it in NodeJS programs, a module should be imported: **zeromq**.

4.2.-MESSAGES

Messages are **what is sent**, so there isn't a framing problem for the app as the buffering is taken care of this problem. They can be "**multi-part**" or "multi-segment", thus ZeroMQ uses a simple structuring support for messages. They are **atomically delivered**, that is that all parts are delivered, or nothing is delivered.

Messages send/receive is **asynchronous**. Internally, ZeroMQ moves messages back-and-forth the in-memory queues to the transports. **Connections/reconnections** among peers is **automatically handled**.

The **message content** is **transparent** to ZeroMQ and there is **no support for marshalling**, so no format is required. Messages are Blobs to ZeroMQ, but ZeroMQ API supports simple string serialization into a message.

You must design your own payload format/protocol. In many cases it may be as simple as just using plain strings. It is possible to use ANY encoding (binary is fine):

- Simple approach: XML messages. Use XML parsers.
- Simpler approach: JSON messages.
- Simplest approach: use each segment for a different piece of information, encoded its own way.

4.3.-ZEROMQ API

4.3.1.-SOCKETS

Creating a socket is simple:

```
const zmq = require('zeromq')

const zsock = zmq.socket(<SOCKET TYPE>)
```

Where <SOCKET TYPE> is one of:

req	push	pub
rep	pull	sub
dealer	*pair*	*xsub*
router		*xpub*

What types to use will depend on the connection patterns they intervene in.

Steps for **establishing communication paths**:

1. One process performs a bind.
2. Other processes perform connect.
3. When done, close.

Bind/connect are decoupled: no ordering requirements.

Multiple connections are possible.

Sockets have message queues associated:

- **Incoming queues**, to hold messages from connected peers. They raise the “message” event when holding some message.
- **Outgoing queues**, holding messages to be sent to peers. Where messages sent from the application are held.

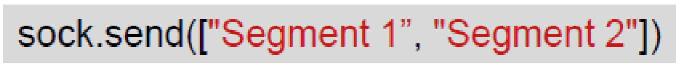
Router sockets keep one pair of outgoing/incoming queues per connected peer. The rest of the sockets do not distinguish among peers. Pub sockets fall out of this discussion.

Pull and sub sockets only set up an **incoming queue**, and, **push and pub sockets** only have **outgoing queues**.

When to **bind** and when to **connect**, in most of the times is indifferent. Nevertheless, there are some observations to point out:

- **All peers** must **meet** at an **endpoint**.
- Endpoints are referred to by their URL.
- For TCP transport:
  - The IP address must belong to one of the bind socket’s interfaces. The bind socket only needs local IP configuration (or none). It does not need to know where the other peers are.
  - The connect socket needs to know where the binding socket is (IP).
- For IPC transport:
  - Inter Process Communication (Unix sockets).
  - URL: ipc://<path-to-socket> (sock.bind("ipc:///tmp/myapp")).
  - Need rw (read & write) permissions over socket at <path-to-socket>.

Segments can be extracted in one call from an array:



```
sock.send(["Segment 1”, "Segment 2"])
```

Segments must be buffers or strings. Strings are converted to buffers, using UTF8 encoding. Non-strings are first converted to strings.

Receive messages is based on “**message**” **events** on the socket. **Arguments** of handler contain the segments of the message.

Sockets have many options, the two most important ones are: **identity** and **subscribe**. **Identity** is convenient when connecting to router sockets:

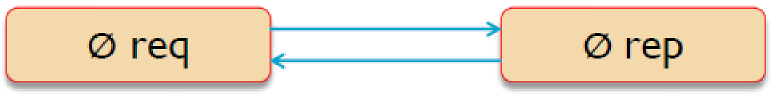
- Sets the ID of a connecting peer to the router.
- It should be set before invoking the connect() method.

Subscribe is used by sub sockets. It sets the prefix filter applied by the pub socket.

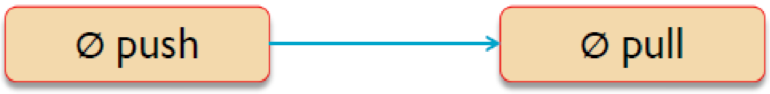
4.3.2.-BASIC PATTERNS OF COMMUNICATION

Basic patterns:

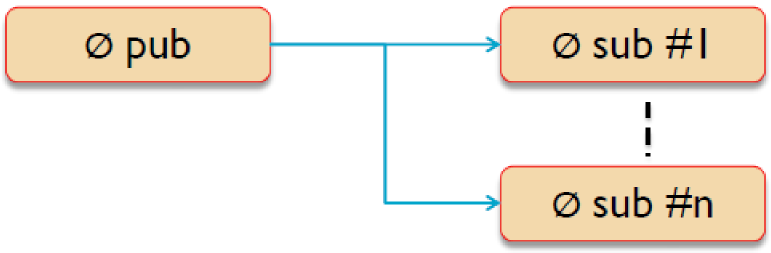
- **Request/reply (synchronous)**:



- **Push-pull**:



- **Pub-sub**:



4.3.2.1.-REQUEST/REPLY

They are implemented using **req** sockets for the client and **rep** sockets for the server.Each message sent via **req** needs to be matched with a corresponding reply from the **rep** socket of the server.

It is a **synchronous** communication pattern. All request/reply pairs are totally ordered. Endpoints can react asynchronously, though.

When a message has been sent through a **req** socket sending a new message through that socket queues it locally until the response message is received. Then the waiting message is sent.

When a request has been received through a **rep** socket, the arrival of a new request through that socket gets it queued until the first request is answered. Then the waiting message is delivered.

Each reply message sent beforehand through a **rep** socket remains enqueued until its associated request is received. When such request arrives, that reply propagation is resumed.

When a **req** connects to more than one **rep**, each request message is sent to a different rep. It follows a round-robin policy. The operation continues being synchronous:

- ZeroMQ will not send new requests until each reply is received.
- No parallelization of requests.

It is also possible to have multiple requesters. It is the typical set-up for a server. **Rep** socket fairly queues incoming messages, so no **req** socket is starved.

When a failure appears at the **rep** socket, **req** is not able to recover, so it needs to be closed, and a new connection has to be established.

Messages exchanged have a first segment empty, referred to as the delimiter. The req socket adds it, without app intervention. The rep socket removes it before handing it to the application, but it adds it again on the reply. The req socket removes it from the reply.

4.3.2.2.-PUSH/PULL

This pattern has a **unidirectional data distribution**. Sender does not expect a reply back, so **messages** do **not wait** for **replies** (concurrent sending). It also accepts multiple connections.

4.3.2.3.-PUBLISH/SUBSCRIBE

This pattern implements **message broadcasting** with a twits: **receivers** can **decide** to **subscribe** to only **some messages**, thus, it is **multicast**. Messages are sent to all connected and available peers. Subscribers can specify filters as prefixes of messages. They can specify several prefixes. Older messages might be lost, if subscriber starts late.