

Mestrado Integrado em Engenharia Informática

Dependable Distributed Systems

Confiabilidade de Sistemas Distribuídos

Project Assignment #1
2021/2022, 2nd SEM


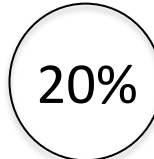
PA#1 Delivery

- Dev.: Start after 25/Mar - Delivery: 9 - 23/April/2022 23h59
- Deadline: 23/April/2022
 - Github Repo (shared w/ “henriquejoaolopesdomingos”)
 - Individual Delivery Form (Google) access by URL to be announced
 - Submission materials
 - Github ProjectRepo (URL) w/ source codes or runtime components, ready for cloning, installation and evaluation
 - README with all necessary indications for configs, installation and deployment
 - Characterization of the delivered solution and checkout of the implemented requirements vs. specified requirements
 - Quick quiz on the concrete implementation and involved background

PA#1: Pre-Requirements

- Programming – Distributed Programming
 - Java (JDK, OpenJDK), Java Debugging
 - JCE, JSSE (Prog. w/ TLS)
 - Tools: EclipseIDE/IntelliJ, java tools (exec. jars)
 - GitHub Project Repo
- C/S, Rest Programming (from DS)
 - Server-Side Rest API
 - Can use Jersey, Spring Boot (spring.io), sping, etc
- Deployment
 - Exec Jars, or Docker, Docker Compose

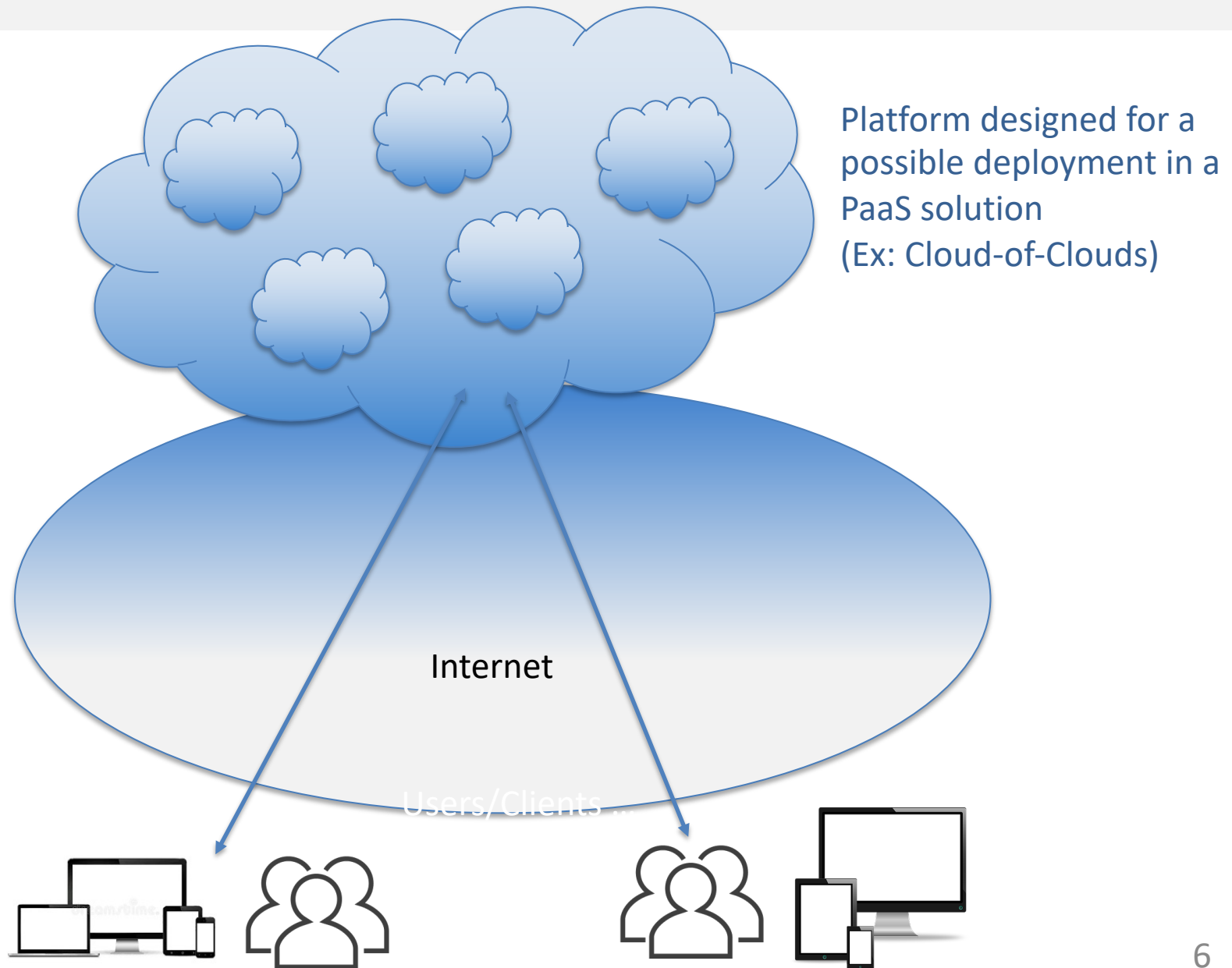
Evaluation criteria

<ul style="list-style-type: none"> Coverage of requirements and related implementation steps Implementation correctness and quality Robustness operation of the delivered solution 	40 % 20% 20%	
<ul style="list-style-type: none"> Delivery form and related quiz questions <ul style="list-style-type: none"> PA#1 implementation characterization Specific implementation-background questions Experimental testing, metrics, critical analysis 	5% 5% 10%	
<ul style="list-style-type: none"> Auto-evaluation / Discussion Discussion/Demo/Rect. (in lab) if/when scheduled Highlights & Highlights Argumentation 	+/- 10% 10% possible extrapoints	
<ul style="list-style-type: none"> Penalizations: -2/20, per delay day 		

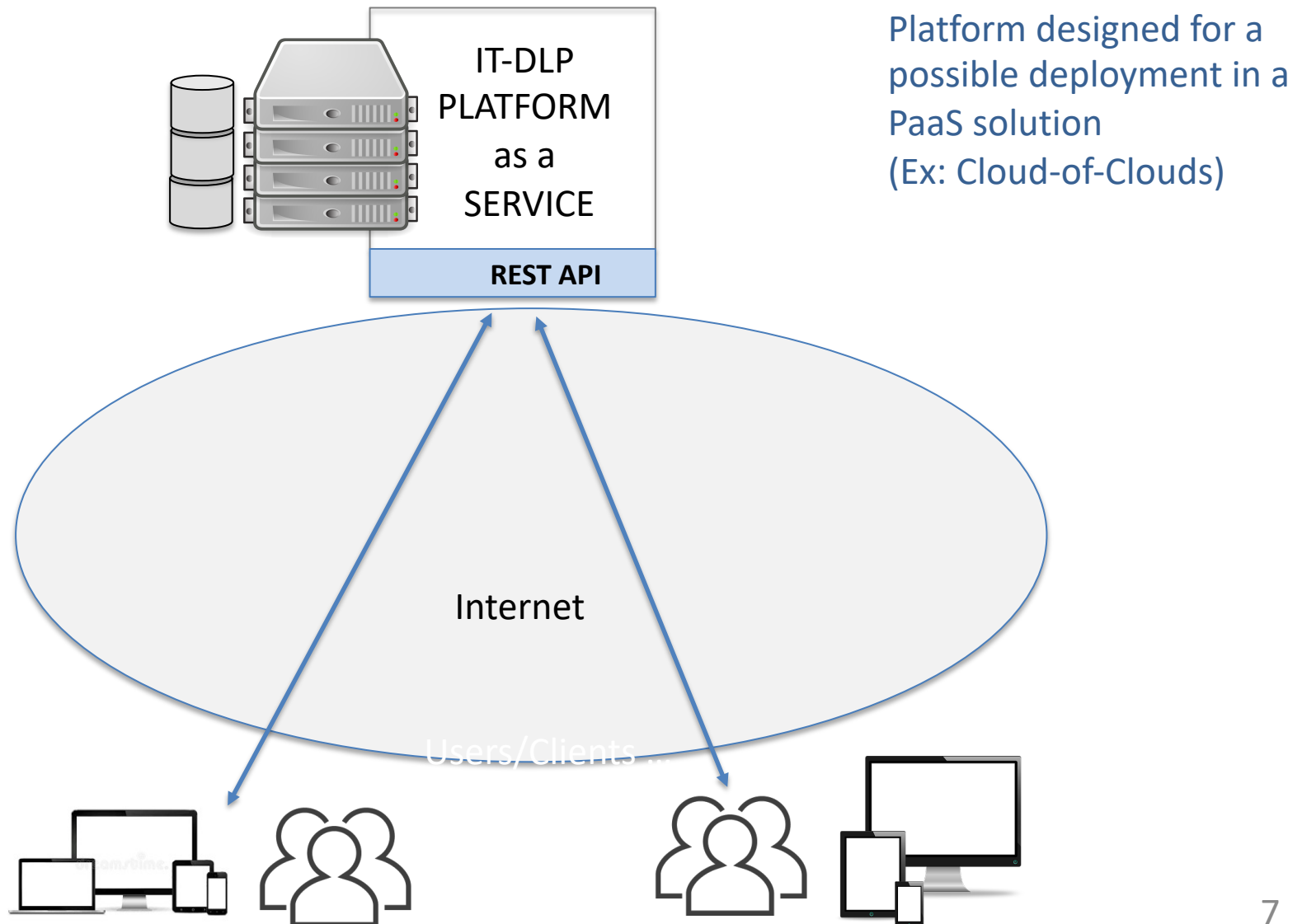
The Project Assignment #1:

Intrusion-Tolerant Decentralized Ledger Platform (IT-DLP)

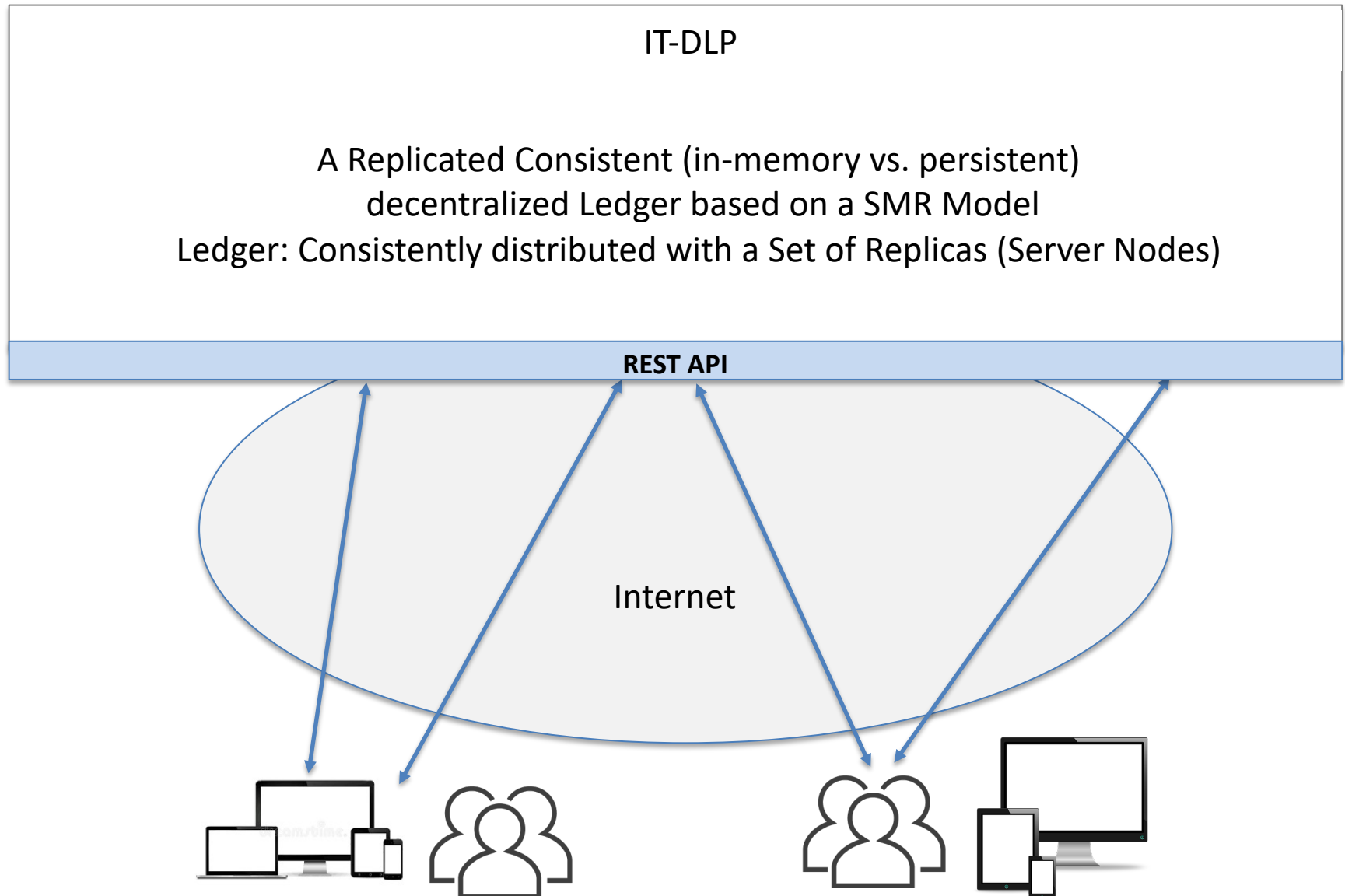
The IT-DLP Platform



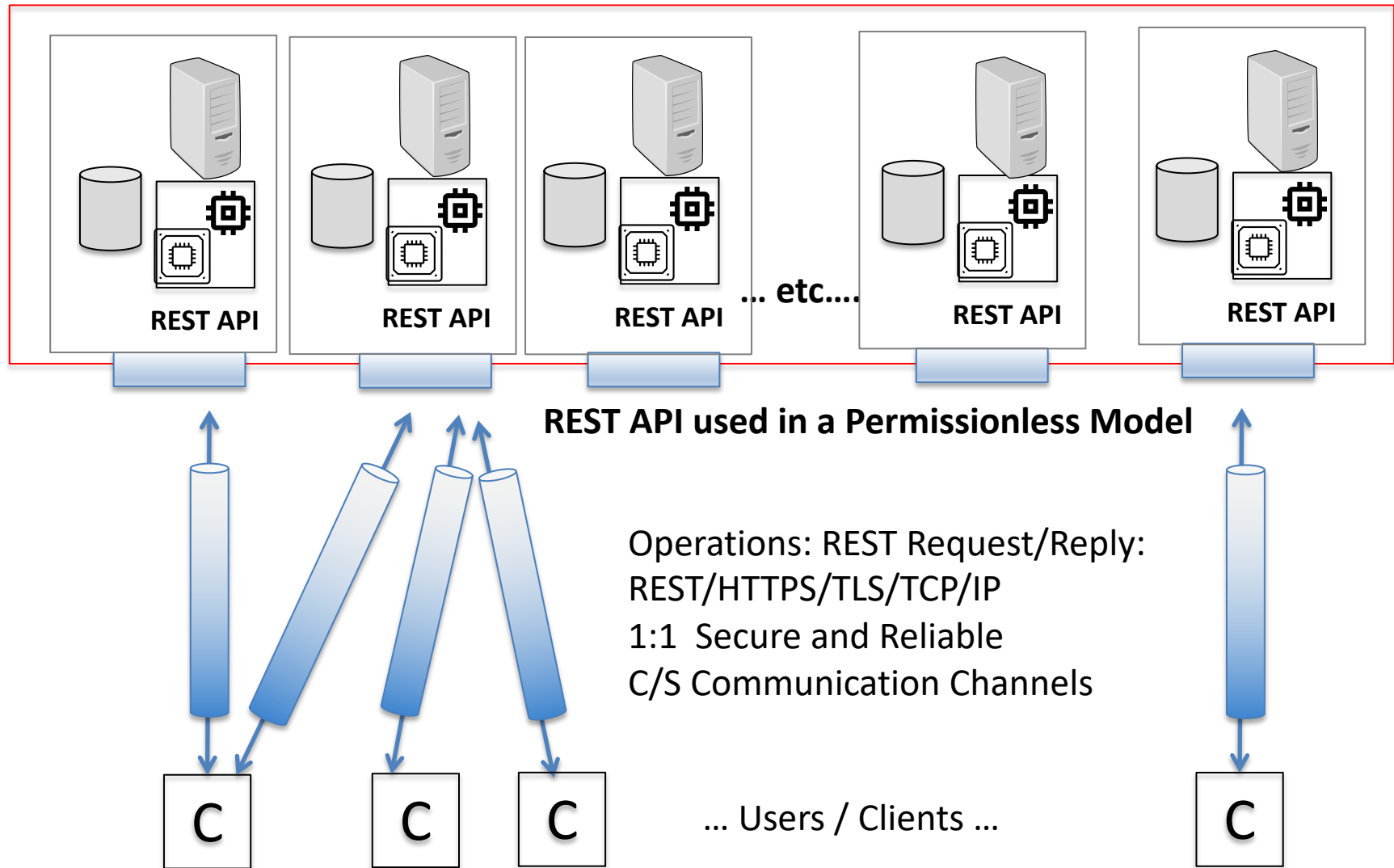
The IT-DLP Platform



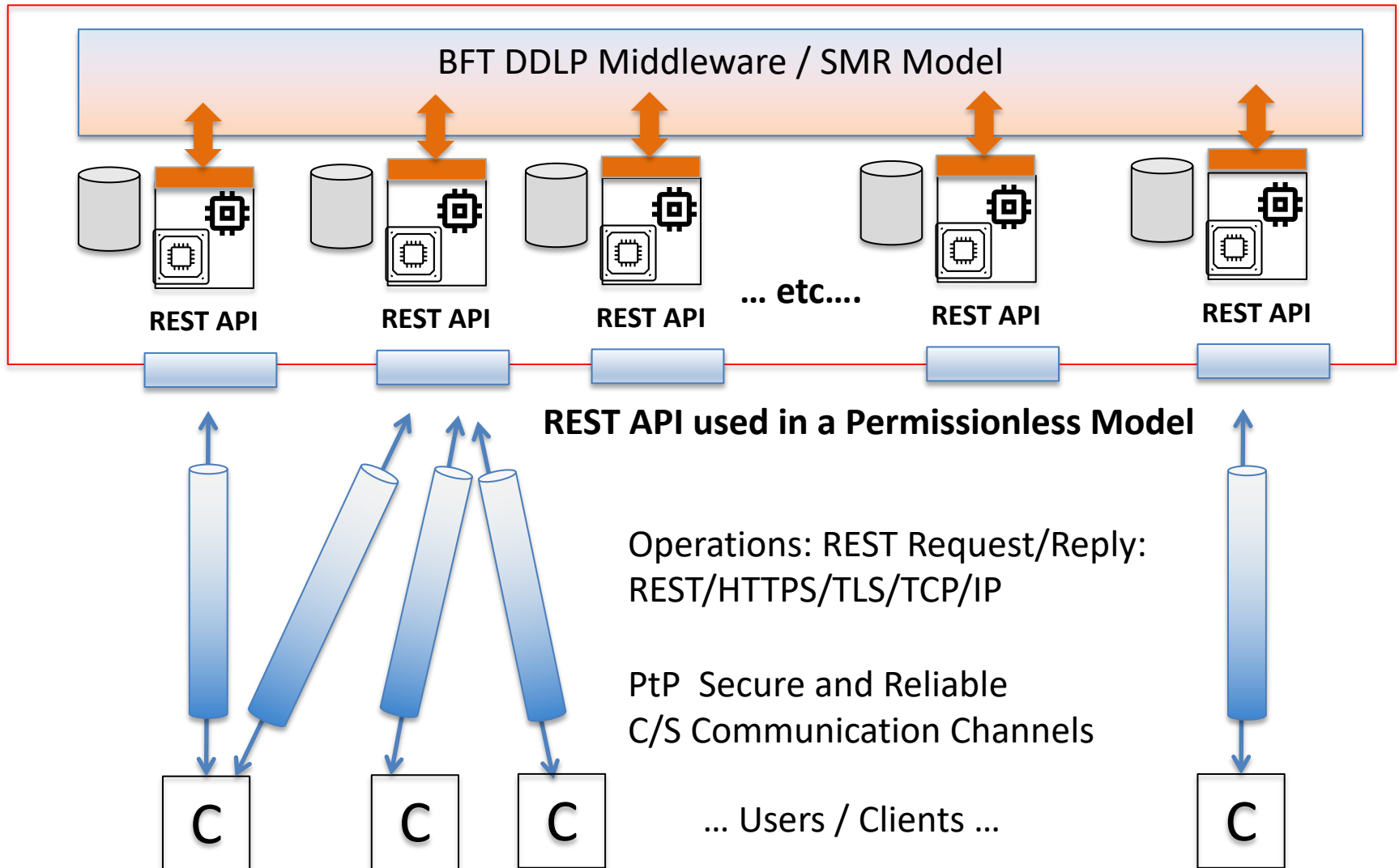
IT-DLP Platform: Scale-Out



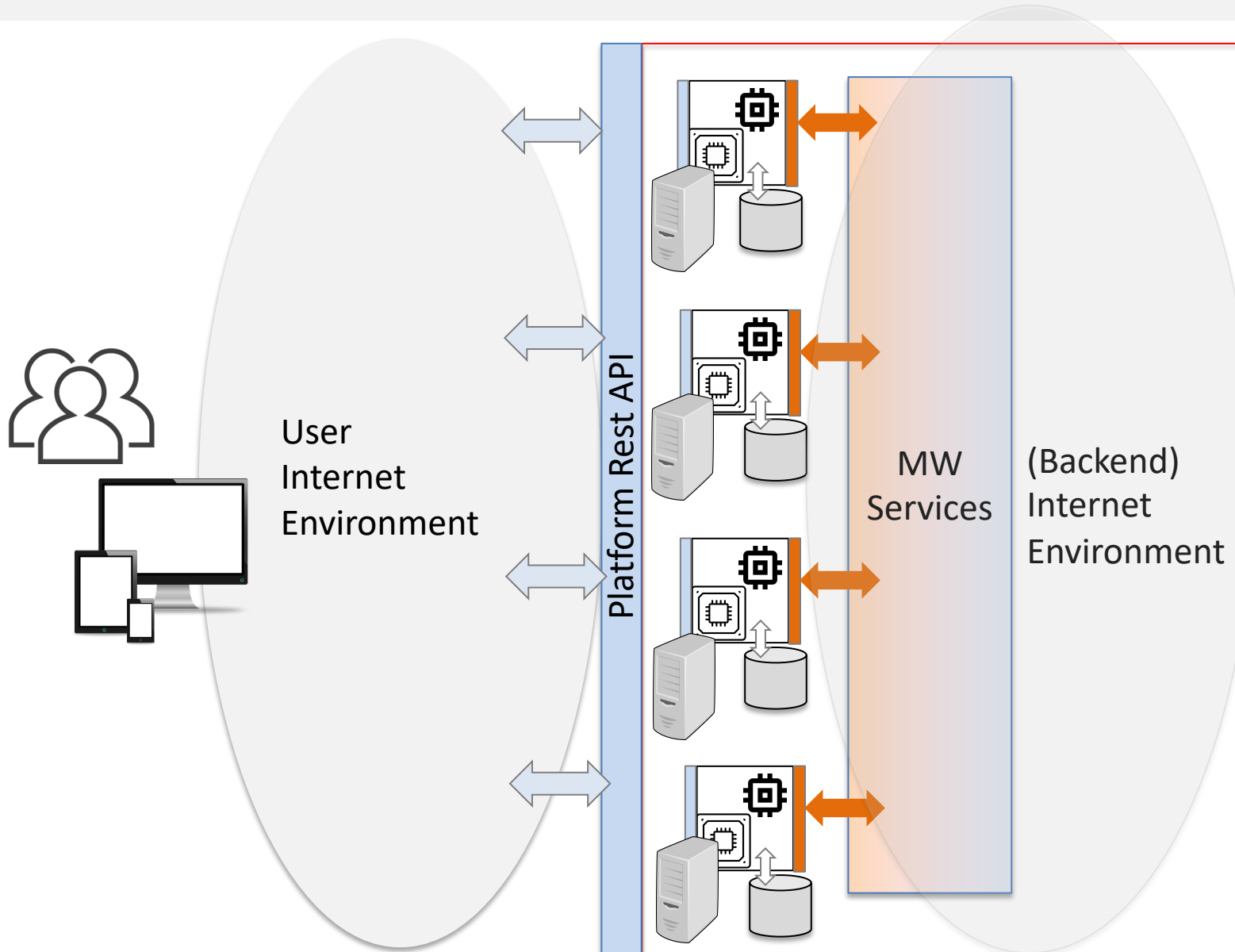
Replicated Service: SMR Model



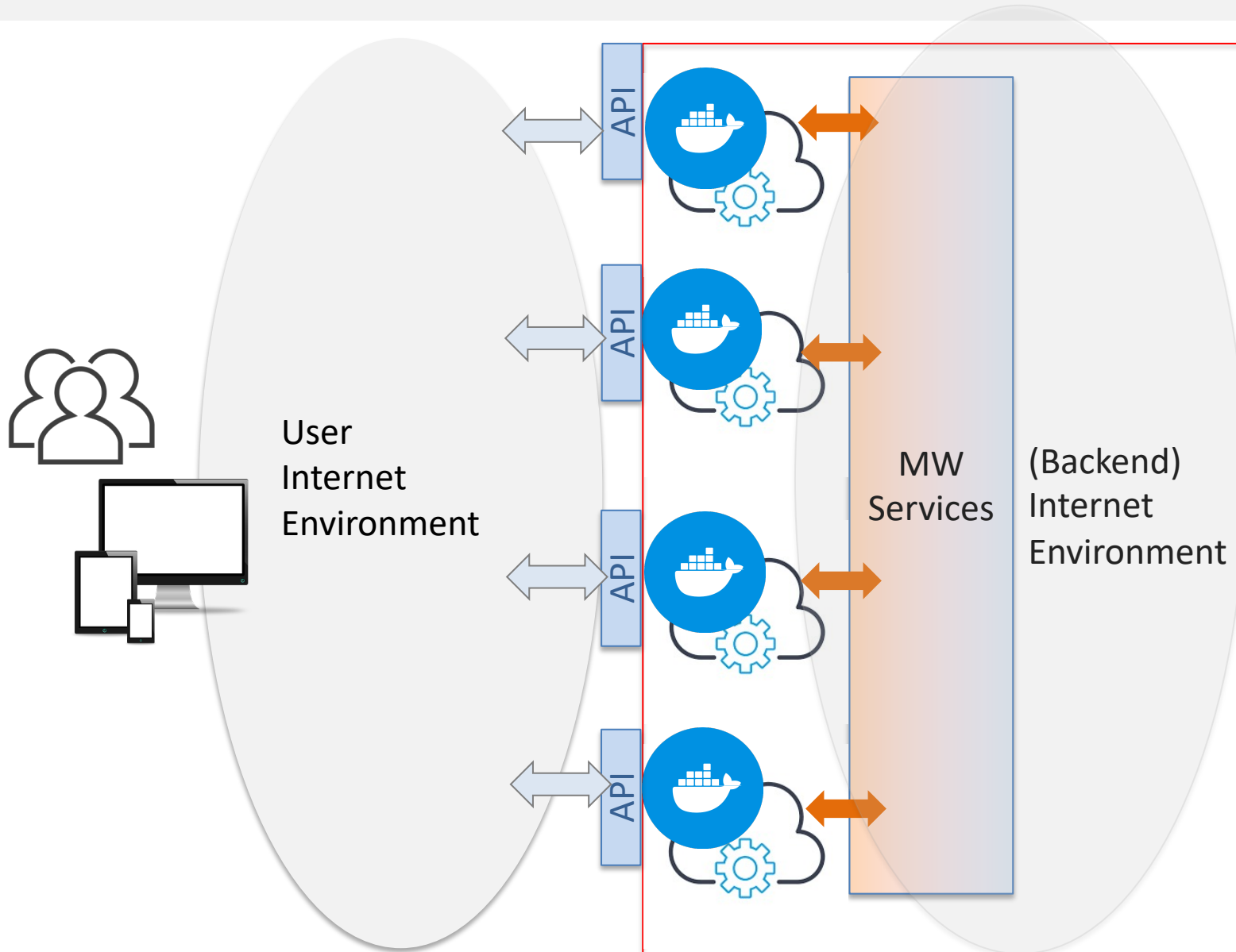
Consistency and Byzantine Fault Tolerance for Intrusion Tolerance



Another perspective ... (a Cloud-Based Cluster)



Virtualization in a Cloud of Clouds: How to ?



Functional requirements

REST API: Operations

Users and Accounts in the Ledger

- The Ledger will be designed as a Permissionless (open) Platform
- Users and UserIDs
 - Users have base external identifiers: Email RFC822
 - $\text{UserID} = \text{SHA256}(\text{email}) || \text{PublicKey}$
or $\text{UserID} = \text{HMAC}_{\text{K-SHA256}}(\text{email}) || \text{PublicKey}$
- Contracts and ContractIDs

Contracts are identified by unique ContractIDs (generated by their user owners)

 - $\text{AccountID} = \text{SHA256}(\text{email} || \text{srn} || \text{timespatml}) || \text{PublicKey}$
or $\text{AccountID} = \text{MAC}_{\text{K-SHA256}}(\text{email} || \text{srn} || \text{timestamp}) || \text{PublicKey}$

This is the reference. You (group) can extend following the base reference !
Discuss and validate your intentions ...

API and operations:

Initial specification

- `createaccount()`
- `getbsmoney()`
- **`sendtransaction()`**
- **`getbalance()`**
- **`getextract()`**
- **`gettotalvalue()`**
- `getglobalvalue()`
- `getledger()`
- (what else in your API ?)

**... that you must address as
the base mandatory
specifications (must have)
but you can (you must)
refine/extend for your own-
specification for
implementation**

... See as an external “view” of transactions in a Blockchain Platform as a Service ;-) ?

Operations: initial specs.

<code>Signed attested.status=createaccount(account, sig)</code>	Operation to create a contract (used as bootstrap operation)
<code>Signed attested.status=loadmoney(account, value, sig)</code>	Operation to load value to a contract (used as bootstrap operation)
<code>Signed attested.status=status=sendtransaction(origin account, destination account, value, sig, nonce)</code>	Send value (from the origin contract) to the destination contract
<code>Signed attested.extract.balance=getbalance(account, sig)</code>	Obtains the current balance of the account in argument, with authentication, and integrity attestation proofs
<code>Signed attested.extract=getextract(account, sig)</code>	Obtains the current extract, with authentication, and integrity attestation proofs
<code>value=gettotalvalue(list of accounts, sig)</code>	Obtains the total amount of value given a set of accounts in the argument
<code>value=getgloballedgervalue(sig)</code>	Obtains the total amount of value registered in in the ledger, corresponding to the <u>total amount</u> of values of all current accounts
<code>signed attested.ledger=getledger()</code>	Obtains the current ledger, with authentication, and integrity attestation proofs

This is the reference. You (group) can extend following the base reference !
Discuss and validate your intentions ...

What guarantees for the required operations ?

- SMR, Total Order Guarantees, under BFT Arguments and BFT Consensus Properties
- Analysis for each one of the required operations
 - Discussion ...

More about the REST API

- Users can have multiple owned accounts
 - only depending on generated keypairs for this purpose
- The operations with gray background are defined and supported only for bootstrap purposes
 - both could be replaced in WA#1 by one single operation with the same effect.
 - Later on (WA#2) these operations can be discarded.
- All operations, supported by REST/HTTPS/TLS invocations
- Parameters or arguments in operations are represented as initial reference
 - You must refine your own specification for datatypes and encodings, as json data types, for your own implementation
 - Note: Signed operations (sig) and signed results

Crypto

- TLS 1.3 (Pref) or at least TLS 1.2: server-side authentication only (and respective X509v3 certificates and required keystores or truststores)
 - Don't need to use certification chains or revocation control (just use truststores for this control)
 - Use only strong *ciphersuites* enabled configs for the TLS
- Digital signatures
 - ECDSA Signatures, EC Keys w/ at least 256 bits, with secure EC curves
 - Client Signed Operations, Signed Results
- For symmetric crypto operation: AES, 256 bit keys, GCM Mode
- For Hash Functions: SHA-256
- For HMACs use constructions with SHA256 and generated MAC keys w/ at least 256 bits

IT-DLP Platform: SW Architecture

What planes ?

Application Support and Integration Plane

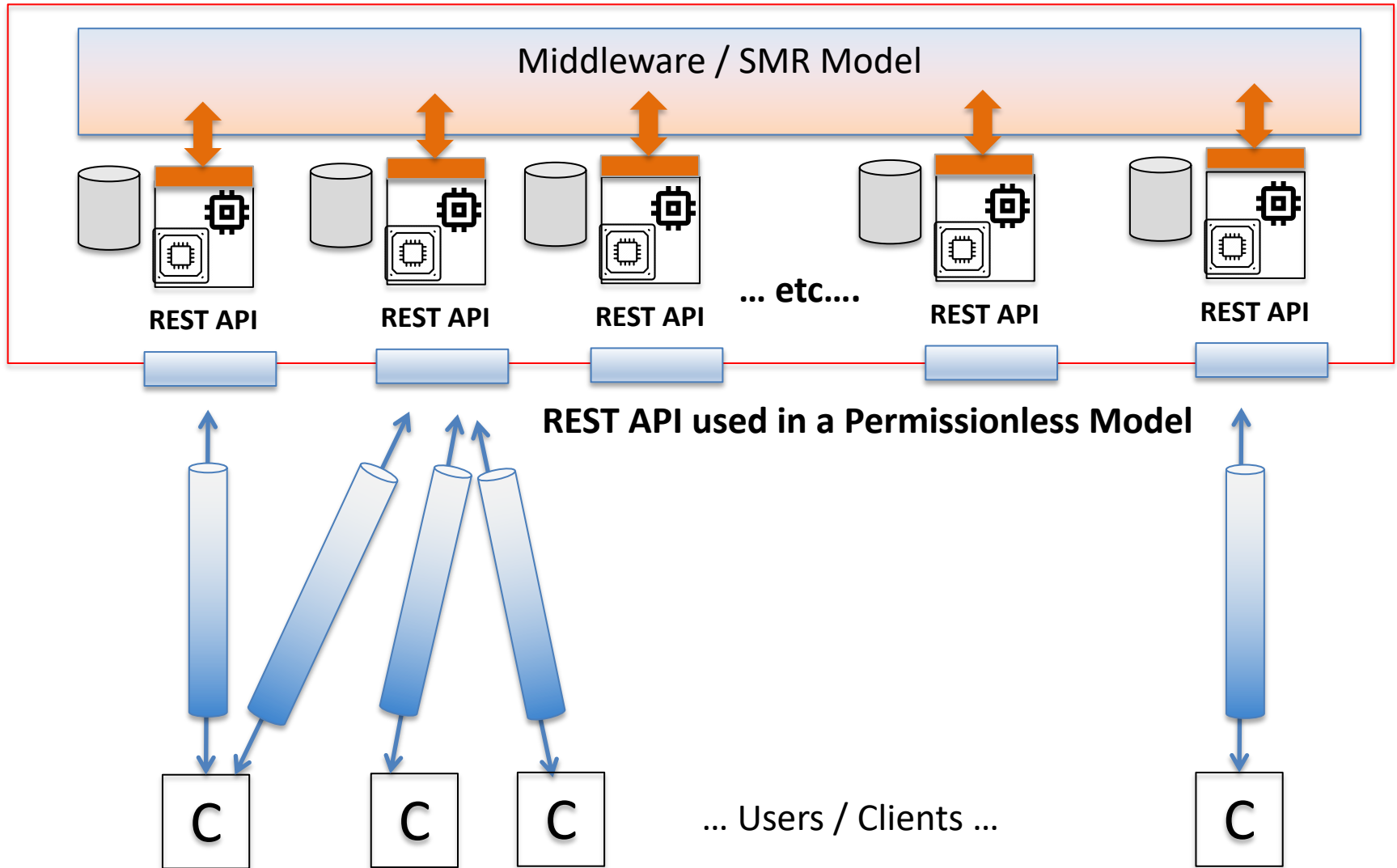
Consensus Plane

Operation Support (or Transactions) Plane

Data Plane

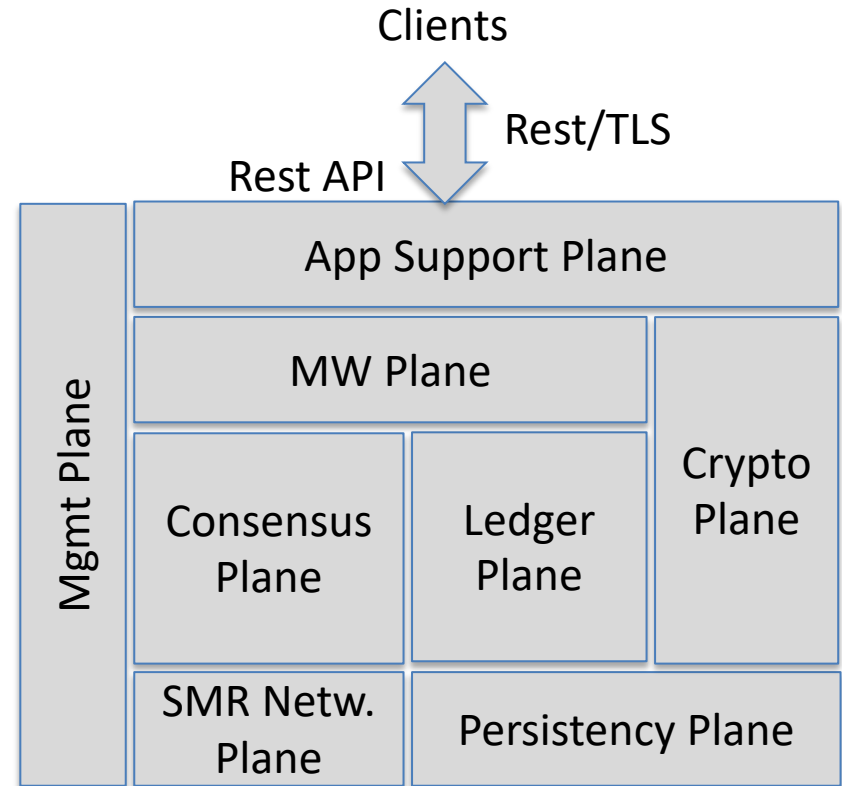
Storage Plane

Platform and Service Planes

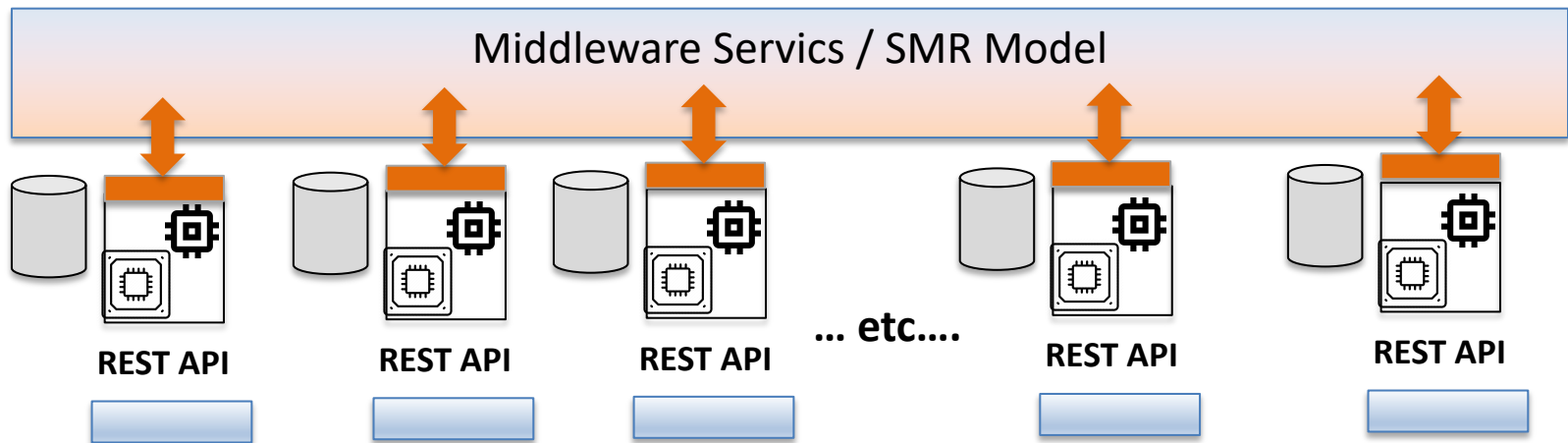


Platform Service Planes

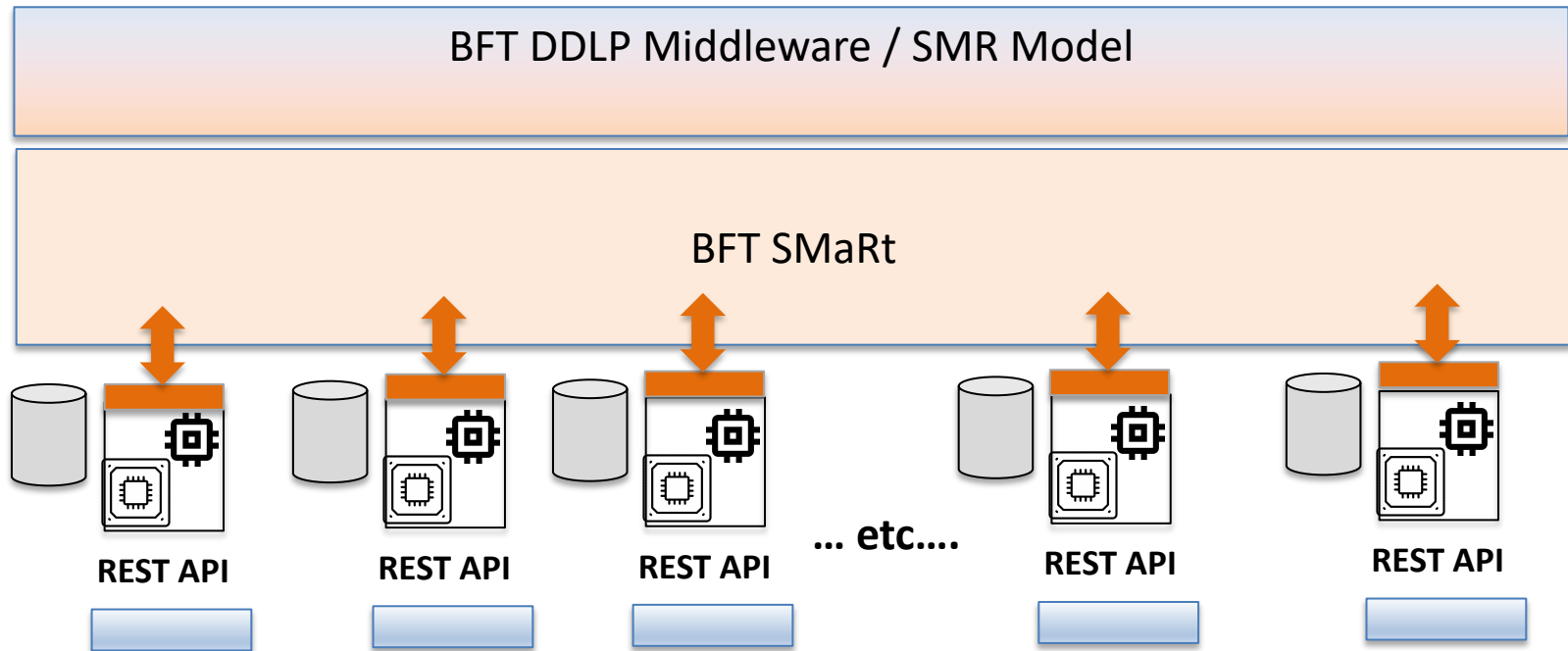
- App Support Plane
 - Rest API and Dispatch. Logics
- MW Plane: MW Services Plane
 - MW Services
- Consensus Plane
 - Consistency
- Crypto Plane
 - Security
- Ledger Plane
 - in Memory Ledger Serv.
- Storage Plane (Persistency Plane)
 - Persistent Ledger State
- (SMR) Network Plane
- Management Plane
 - Configs, setup, MGMT Tools, etc...)



Platform Services:



MW Services / Consensus Plane: leveraged from BFT SMaRt



What is BFT SMaRt ?

- (sic, from authors): A high-performance Byzantine fault-tolerant state machine replication library developed in Java with simplicity and robustness as primary requirements. The main objective is to provide a code base that can be used to build dependable services and also extended to create new protocols, including Intrusion-Tolerance Protocols and Services

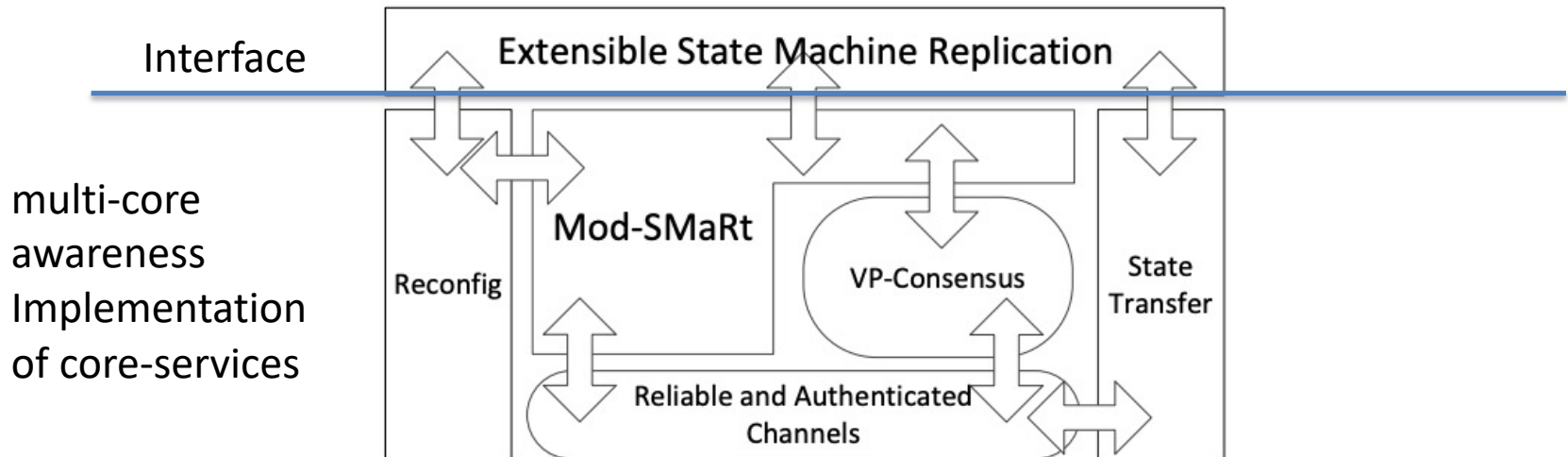
BFT SMaRt

- BFT-SMART: Java based BFT SMR library (and runtime services)
- Targets both high-performance in fault-free executions and correctness if faulty replicas exhibit arbitrary (byzantine) behavior.
 - Possible reconfigurations of the replica set and efficient and transparent support for durable services
 - Processes (replicas) – membership configurations and parameterizations
 - Internal logging for recoverability
 - State-transfer
 - SMR w/ consensus layer supported by PBFT using reliable and secure “broadcasted” operations
 - On top of Authenticated LINKS using HMACs
 - On Top of Authenticated LINKS using TLS

BFT-SMaRt References

- Leveraged from the BFT-SMaRt solution- A (use v1.2)
 - <https://github.com/bft-smart/library>
- BFT SMaRt Paper (ref, DSN 2014)
 - <http://www.di.fc.ul.pt/~bessani/publications/dsn14-bftsmart.pdf>[bft-smart/library](https://github.com/bft-smart/library)
- You have also more info sources here:
 - Technical Report, Getting Started with ... , Setup/Config Parameterizations, some FAQ-Questions,
 - <https://bft-smart.github.io/library/>

What means “complete” in practice ?



BFT arguments: $N \geq 3f + 1$ (for f byzantine replicas)

Core Protocols:

- Total Order Multicast (Mod-SMaRT module) using an underlying consensus primitives on top of a Leader-Driven Byzantine Consensus (PAXOS-Based Model)
 - VP Consensus Layer
- State-Transfer: allowing for repairing and reintegration of replicas in the system, without restarting the whole replicated service
 - Recoverable persistent states (durable logging)
- Reconfiguration (possible dynamic view-membership management to add/remove replicas)
- Base communication leveraged by TLS-enabled reliable and authenticated channels

Setup and configuration mechanisms

Failure model

- The system can be configured to support two other fault models.
- Crash Fault Model: $f < n/2$ (simple minority) , or $n \geq 2f+1$
- Byzantine Fault Tolerance Model; $n \geq 3f+1$, or $f < n/3$

Flexibility: configurations (network plane):

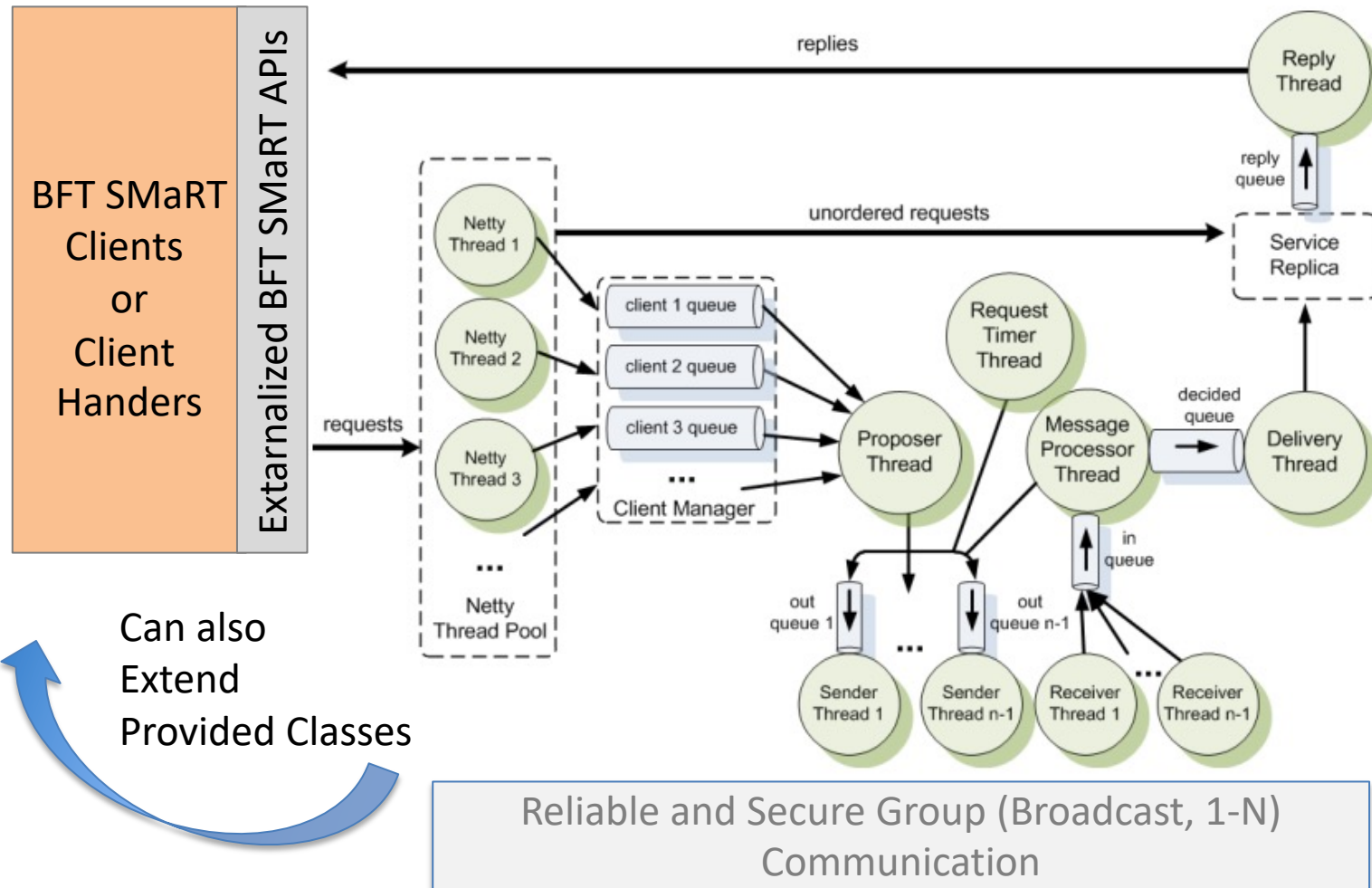
- Can use authenticated FIFO reliable channels
 - Using MACs
 - Using TLS

Other configurations:

- <https://github.com/bft-smart/library/wiki/BFT-SMaRt-Configuration>

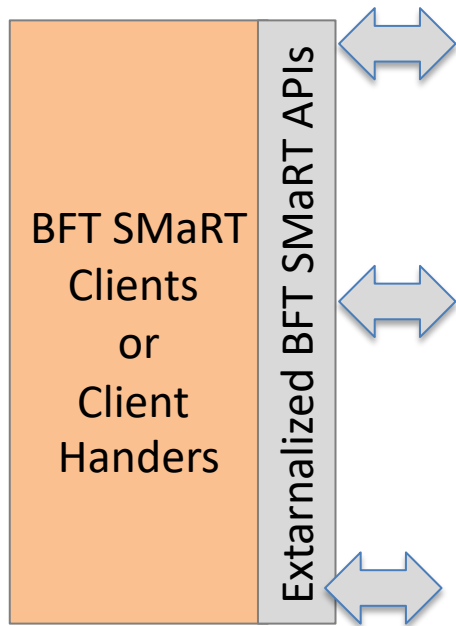
Implementation

- 13.5K lines of Java code ~more than 90 files
- Internal architecture



BFT SMaRt Interface

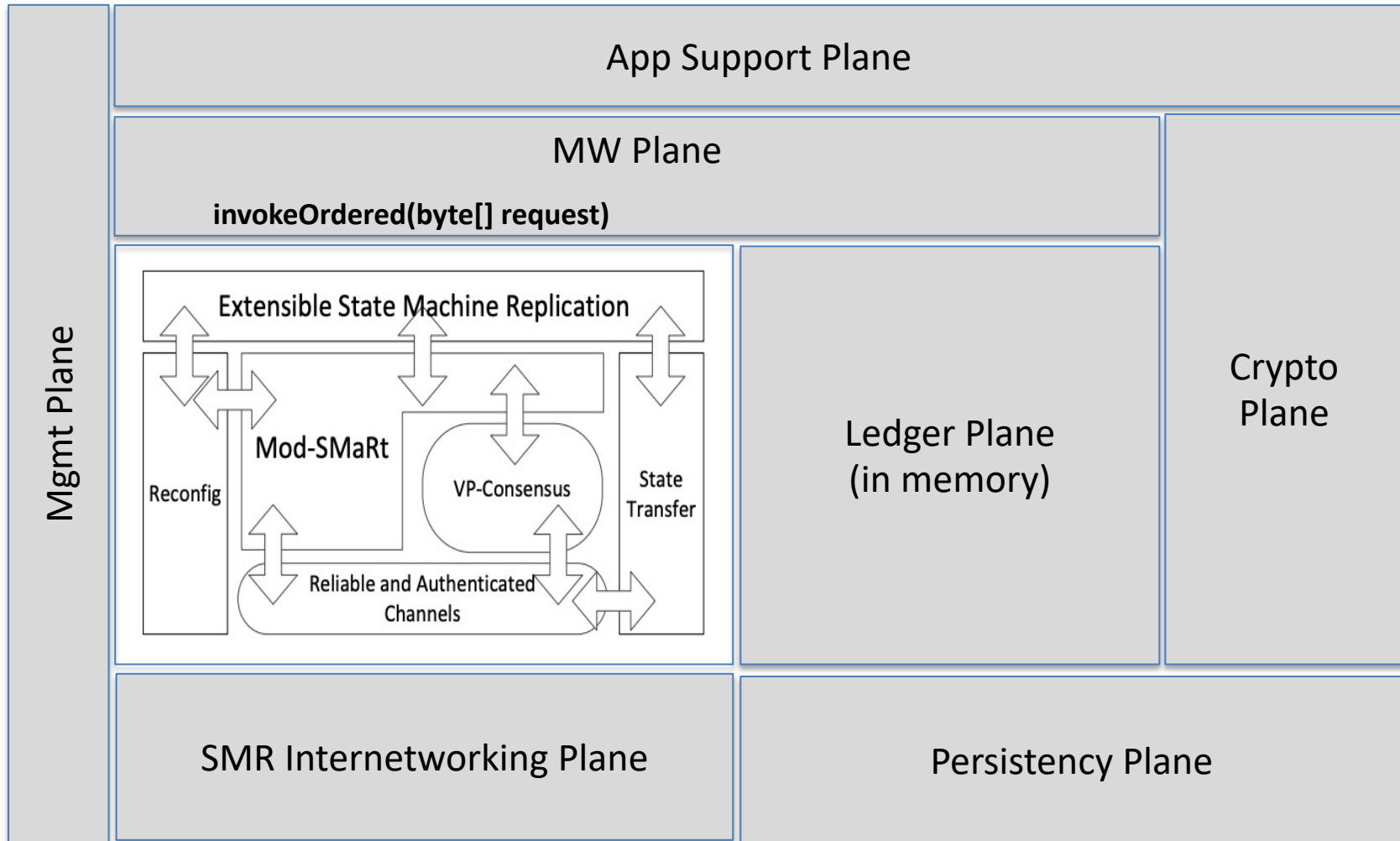
Client Side API



Client (Main Externalization) API:

- Synchronous (blocking) invocations
invokeOrdered(byte[] request)
- Synchronous (not blocking) invocations
invokeUnordered(byte[] request)
invokeUnorderedHashed(byte[] request)
- Asynchronous (not blocking) invocations
invokeAsynchRequest(byte[] request,
ReplyListener replyListener,
TOMMessageType reqType)

Platform Planes w/ BFT-SMaRt Integration



Programming Environment and Tools

PA#1 and later for PA#2

What about the ledger persistency ?

- You will need a storage solution for persistency. Can choose among different solutions:
 - RAW-storage (object serialization / file-system)
 - REDIS
 - Other Key-Value Store (In-Memory vs. InDisk or Both)
 - MongoDB
 - Hybernate
 - Apache Hbase
 - Apache CouchDB
 - What else is better, more appropriate from your previous practical skills ?

Methodology (discussion)

A recommendation for methodology: design, development, test and critical analysis

Can go Step by Step until the delivery date ...

- Delivery initially planned for

1) Try to start with a centralized version

REST API, Client/Server and an initial “in memory” data structure for the Ledger, to maintain the necessary logged operations and their elements (can start without REST/TLS) but implementing the operations in the REST API from a stabilized specification from the base requirements and indications.

- Test the latency and throughput of operations w/ a micro testbench and workload.
- Test with concurrent client workloads

2) Integrate the BFT Smart to have a replicated Ledger solution to build your consistency SMR plane. Test !

Synchronous (invokeOrdered(byte[] requests)

3) Add TLS support for client-requested operations (with the necessary configs, parameterizations and testing). Test !

4) Review/Refine/Optimize the design with an appropriate data-structure for the ledger and required elements in order to have a better support for the client operations.

- Test the latency and throughput of operations w/ a micro testbench and workload.
- Test with concurrent client workloads

5) Add the persistency component (to have now the ledger plane with “in-memory” and “in-disk” support)

6) Revise everything and analyze the achieved BFT arguments of your solution ! (Think as an adversary) !

6) Deploy for WA1 delivery

- Highlighted option: “A Dockerized” version for your DDLP solution. Test !
- Test everything and prepare the delivery reading and answer the delivery form and indications – there will be a verification test for client workload and testing to evaluate latency and throughput measurements that you will report on the delivery. Test everything !