Antidote

A causally-consistent cloud store for high scalability, availability and low latency

Setup for this tutorial

- For the hands-on parts, you need to load:

```
docker run -t -i cmeiklejohn/antidote-tutorial
```

- *Warning*: This will take a while….

Antidote in a nutshell

Cloud-scale database

- Performance: sharded, parallel DC
- Widely geo-replicated:
  - ➡ Many DCs, large or small, core or edge
- Available: updates do not block
  - ➡ Enables fast response
- Sweet spot: performance vs. usability

Reference platform for EU project **SyncFree**:
Large-scale computation without synchronization

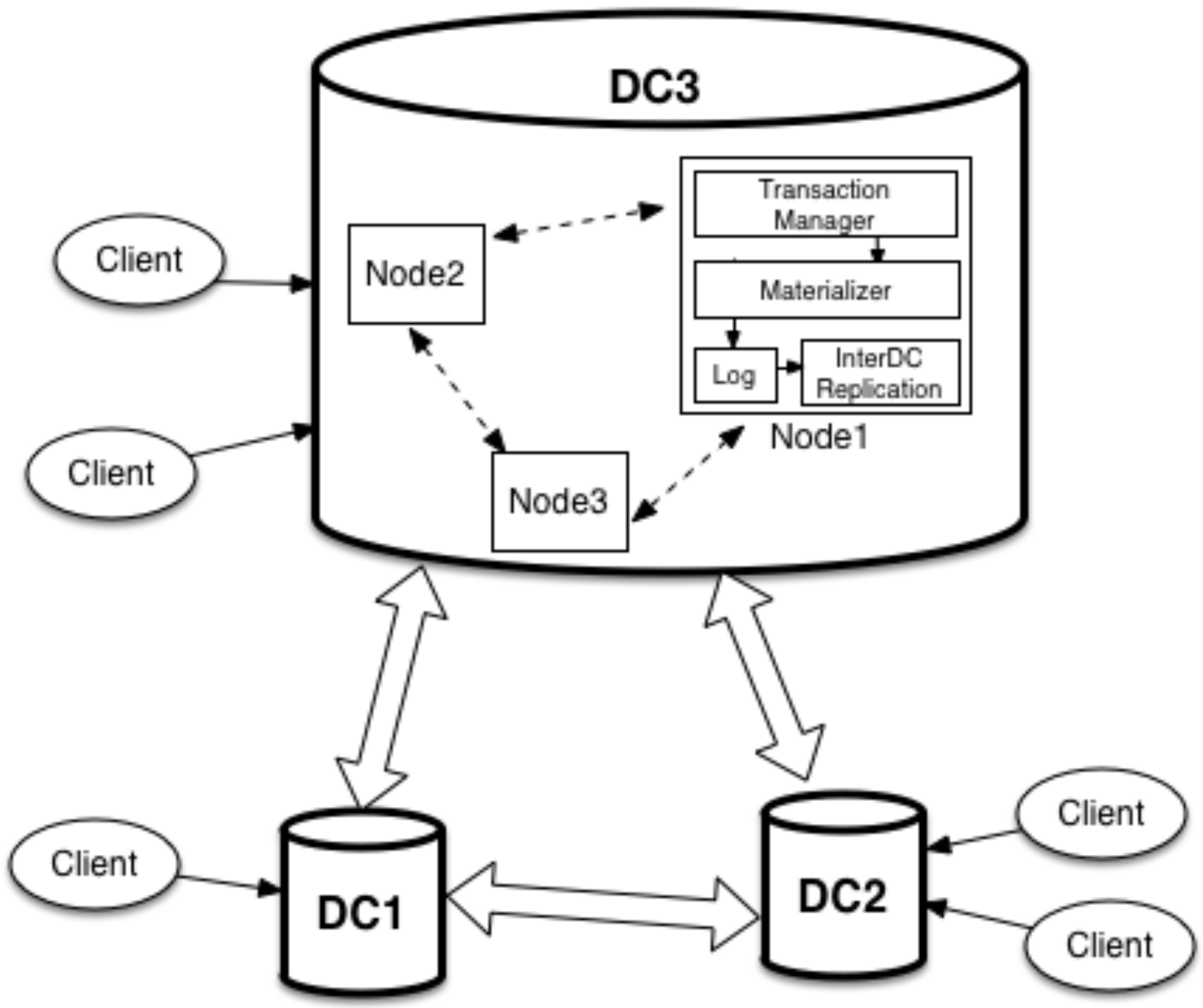
Outline

- Part I: Architecture
- Part II: CRTDs
- Part III: Transactions
- Part IV: More cool stuff

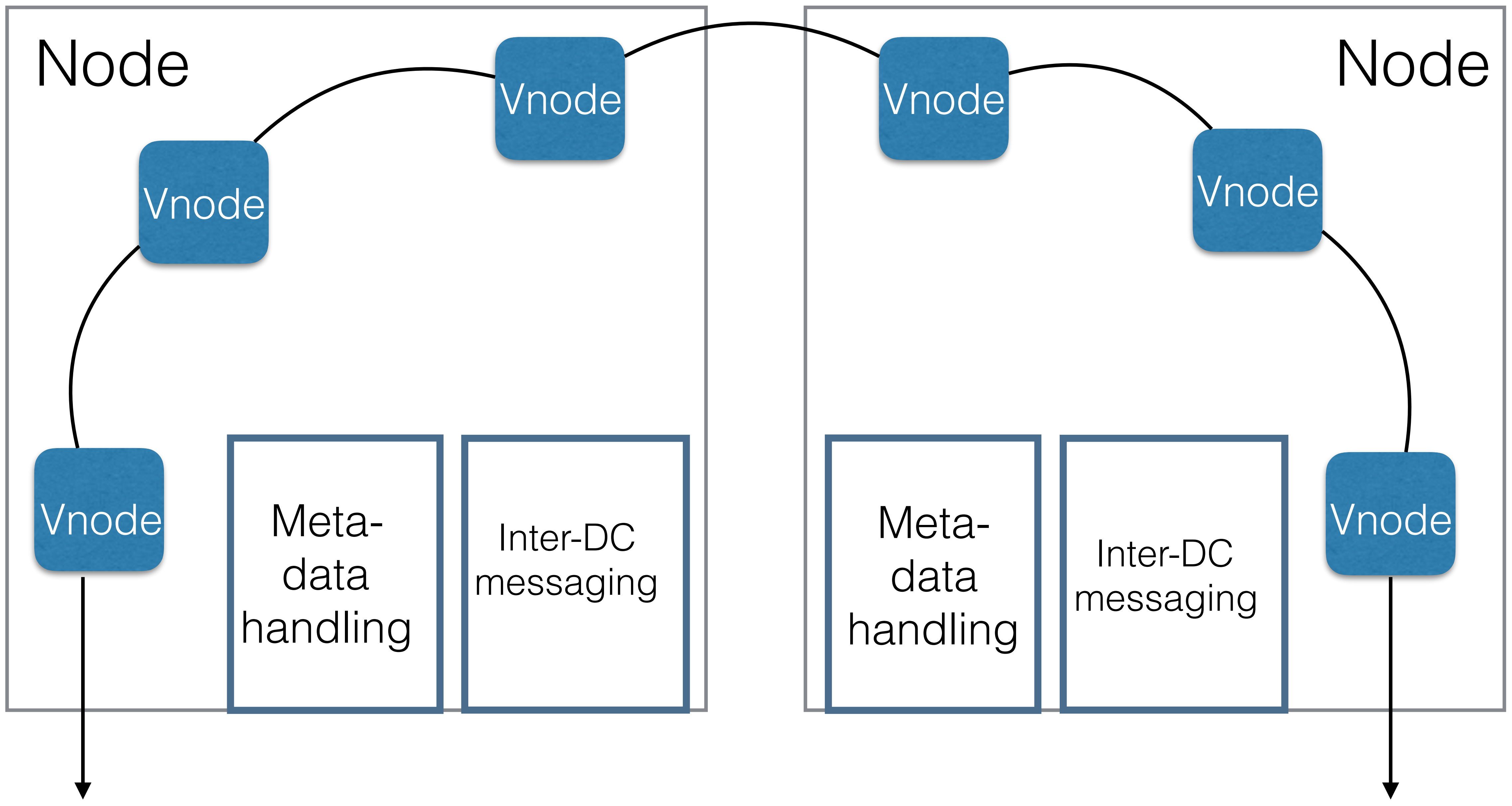
Part I

Architecture

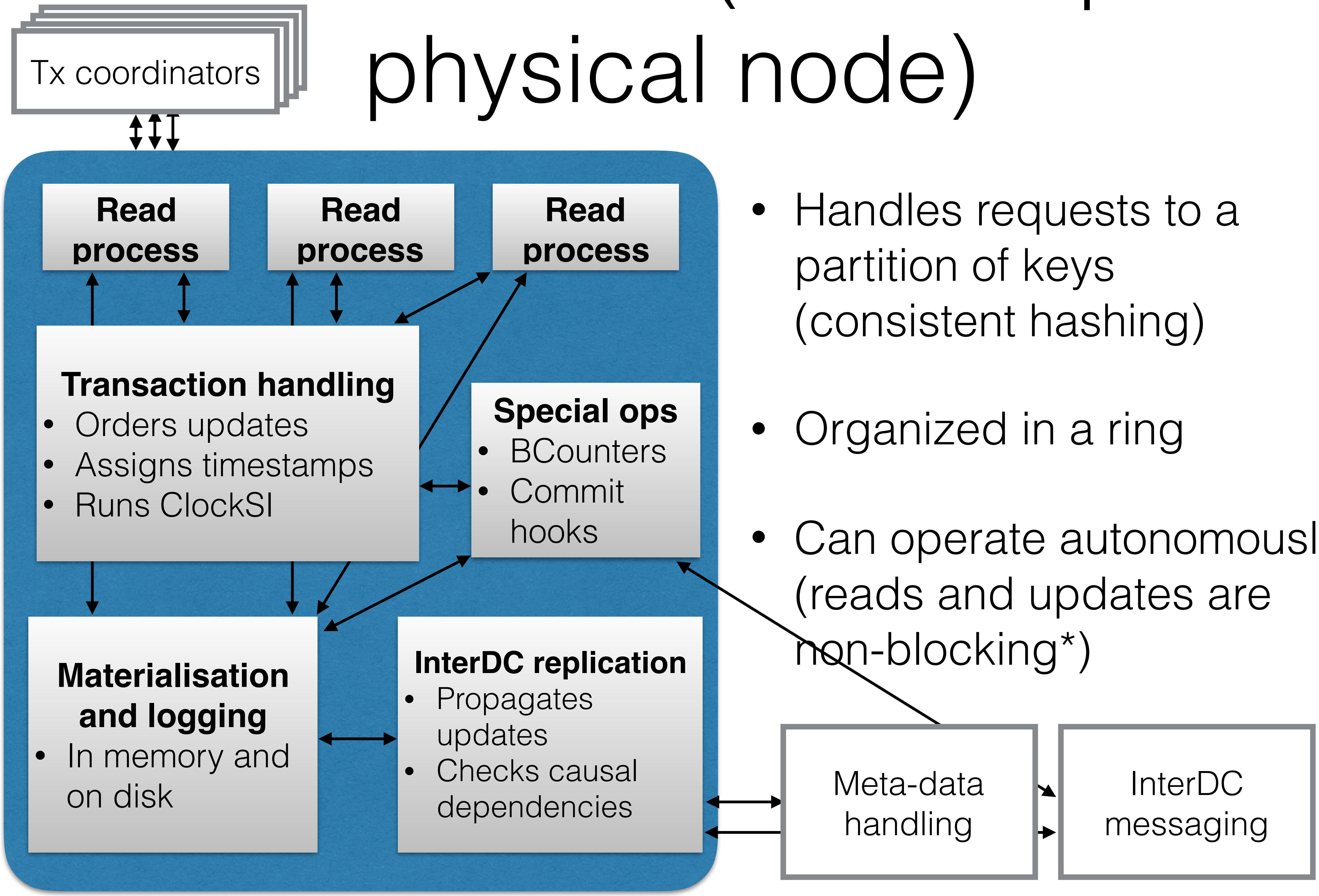
Architecture



(Physical) Nodes



Virtual Nodes (Several per physical node)



- Handles requests to a partition of keys (consistent hashing)
- Organized in a ring
- Can operate autonomously (reads and updates are non-blocking*)

Use case: MyBooks app

- Maintain book lists for users that swap their books
  - ➡ User profile (name, email)
  - ➡ List of owned books
  - ➡ List of borrowed books

Part II

CRDTs

Conflict-free replicated data types

Data type

- Encapsulates state
- Well-defined interface

Replicated

- At multiple nodes (even clients!)

Conflict-free

- Update replica without coordination
- Convergence guaranteed (formal properties)
- Decentralized
- **No lost updates**

CRDTs in Antidote

Counter	`{increment, integer()}`
	`{decrement, integer()}`

ORSet	`{add, term()}`
	`{remove, term()}`
	`{add_all, [term()]}`
	`{remove_all, [term()]}`

GSet*	`{add, {term(), actor()}}`
	`{add_all, {[term()], actor()}}`

LWW Register*	`{assign, {term(), non_neg_integer()}}`
	`{assign, term()}.`

MV Register	`{assign, {term(), non_neg_integer()}}`
	`{assign, term()}`

Map*	`{update, {[map_field_update() | map_field_remove()],`
	`actorordot()}}.`
	
	`-type actorordot() :: riak_dt:actor() | riak_dt:dot().`
	`-type map_field_remove() ::  {remove, field()}.`
	`-type map_field_update() :: {update, field(), crdt_op()}.`
	`-type crdt_op() :: term(). %% Valid riak_dt updates`
	`-type field() :: term()`

RGA
(replicated growable array)
	`{addRight, {any(), non_neg_integer()}}`
	`{remove, non_neg_integer()}`

Read and update

```
type bound_object() = {key(), crdt_type(), bucket()}.
type snapshot_time() = vectorclock() | ignore.

update_objects(snapshot_time(), properties(),
    [{bound_object(), operation(), op_param()}]) ->
         {ok, vectorclock()} | {error, reason()}.

read_objects(snapshot_time(), properties(),
     [bound_object()]) ->
          {ok, [term()], vectorclock()}.
```

MVReg CRDT

MV Register	`{assign, term()}`
	`{assign, {term(), non_neg_integer()}}`

- Multi-Value Register
- Assign only allows to set one value
- Concurrent assignments can lead to multiple values being set; conflict is propagated to the user
- If no timestamp is provided, entry that is associated the actor is incremented and this timestamp will be used. Explicit times useful when client updates to different replicas

Hands-On

- Add user information for Michel

```
User1 = {michel, antidote_crdt_mvreg, user_bucket},

{ok, Time2} = antidote:update_objects(ignore, [],
    [{User1, assign, {["Michel", „michel@blub.org"],
    client1}}]),

{ok, Result, Time2} = antidote:read_objects(ignore, [], [User1])
```

- Add another user „Alex, alex@wow.com“

- Change user information (e.g. add surname name)
  Read modified information

OR-Set CRDT

ORSet	`{add, term()}`
	`{remove, term()}`
	`{add_all, [term()]}`
	`{remove_all, [term()]}`

- Observed-remove Set
- Only elements that have been present in the version on which the update is executed, will be removed
- Elements can later be re-added

Hands-On

- Add books to list of owned books

```
Owned = {michel, antidote_crdt_orset, owned_bucket},

{ok, _} = antidote:update_objects(ignore, [],
              [{Owned, add, "Algorithms"}]),
{ok, _} = antidote:update_objects(ignore, [],
              [{Owned, add_all, ["Erlang","Java"]}]),

{ok, Result, _} = antidote:read_objects(ignore, [],
              [Owned])
```

- Remove Java book and check the result!

Part III

Transactions

Highly Available Transactions

Transaction with weak isolation

- Consistent snapshot reads
- All-or-nothing writes

No synchronization: available, fast

Compatible with

- Causal consistency
- CRDTs

Why transactions?

- Weak invariant preservation
- Example: Maintaining friend lists
  - *friendOf(x,y)* ⇔ *friendOf(y,x)*
  - foreign key constraints
- Restrictions!

Transaction API

```
type bound_object() = {key(), crdt_type(), bucket()}.
type snapshot_time() = vectorclock() | ignore.

start_transation(snapshot_time(), properties()) ->
         {ok, txid()} | {error, term()}.

update_objects([{bound_object(),operation(), op_param()}],txid()) ->
         ok | {error, term()}.

read_objects([bound_object()], txid()) -> {ok, [term()]}.

commit_transaction(txid()) ->
         {ok, vectorclock()} | {error, term()}.
```

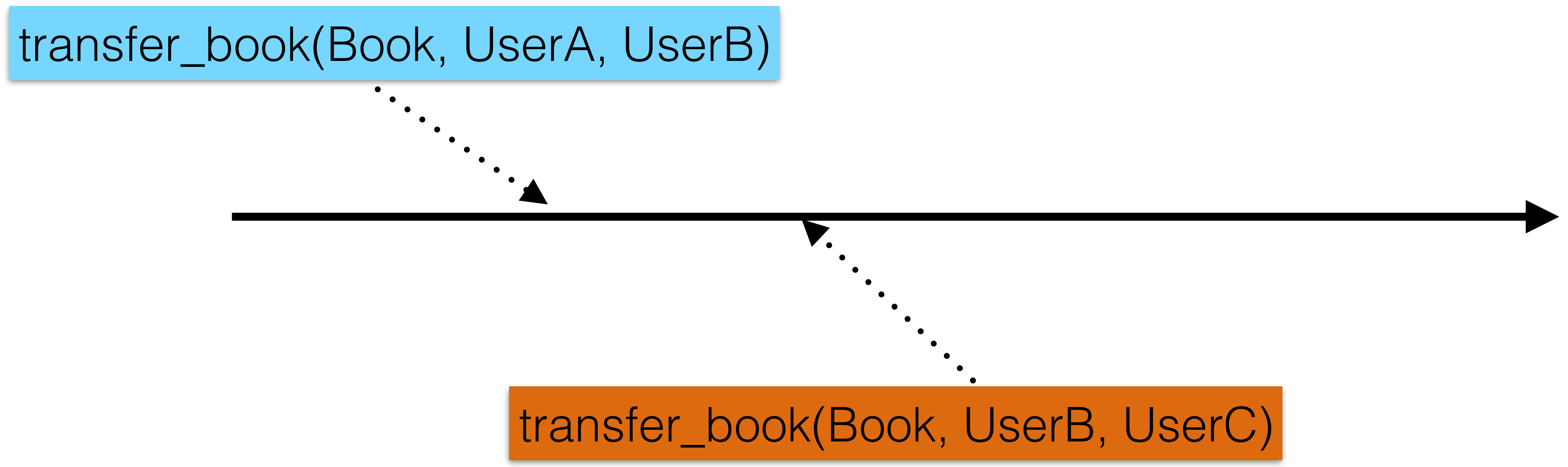
Hands-On

- Michel lends book to Alex

```
{ok, TxId} = antidote:start_transaction(ignore, []),

Owned = {michel, antidote_crdt_orset, loan_bucket},
ok = antidote:update_objects([{Owned, add, "Erlang"}], TxId),

Borrowed = {alex, antidote_crdt_orset, borrowed_bucket},
ok = antidote:update_objects([{Borrowed,add,"Erlang"}], TxId),

{ok, TS} = antidote:commit_transaction(TxId),

%% Check
{ok, Result,_} = antidote:read_objects(TS,[],[Borrowed,Owned])
```

- Your try: Alex returns book to Michel

Causal Consistency


transfer_book(Book, UserA, UserB)
transfer_book(Book, UserB, UserC)

- Intuitive
- Guarantees read-your-own-writes, monotonic reads, monotonic writes (optionally strongly ordered per DC)
- **Strongest** partition tolerant and available consistency model
- Causal broadcast of updates required for (op-based) CRDTs

Hands-On

- Michel lends book to Alex, and Alex forwards book to Hugo

```
{ok, TxId} = antidote:start_transaction(ignore, []),
Owned = {michel, antidote_crdt_orset, owned_bucket},
ok = antidote:update_objects([{Owned, remove, "Java"}], TxId),
Borrowed = {alex, antidote_crdt_orset, borrowed_bucket},
ok = antidote:update_objects([{Borrowed, add, "Java"}], TxId),
{ok, TS} = antidote:commit_transaction(TxId),

{ok, TxId2} = antidote:start_transaction(TS, []),
ok = antidote:update_objects([{Borrowed,remove,"Java"}],TxId2),
Borrowed2 = {hugo, antidote_crdt_orset, borrowed_bucket},
ok = antidote:update_objects([{Borrowed2,add,"Java"}], TxId2),
{ok, TS2} = antidote:commit_transaction(TxId2).
```

Stronger consistency

- Let’s add as restriction that a user may not borrow more than 2 books at the same time
- User could try to outsmart the system by connecting to different DCs for their requests
- Need a way maintain this stricter invariant
    - ➡ Bounded counters

Part IV

More cool stuff

Protocol buffer interface

- Uses PB encoding for efficient message transfer
- Connection via protocol buffer socket instead of RPC calls
- Supports working with local obj proxy at client side
- API very similar to what we have seen so far

Commit hooks

- Hooks are functions that are executed when updating an object

```
fun (update_object()) -> {ok, update_object()} | {error, Reason}.
type update_object() :: { {key(), bucket()}, crdt_type(), update_op() }
type update_op() :: {atom(), term()}
```

- Pre- / Post-commit hooks can be registered per bucket
- Before an object in the bucket is updated, pre-hook might modify the update operation
- Post-hook gets the (potentially modified) operation and executes before returning to client

```
register_post_hook(bucket(), module_name(), function_name())
     -> ok | {error, function_not_exported}.

register_pre_hook(bucket(), module_name(), function_name())
     -> ok | {error, function_not_exported}.

unregister_hook(pre_commit | post_commit, bucket()) -> ok.
```

Transaction protocols

- Cure protocol to define snapshots and causal dependencies, geo-replicated version of ClockSI protocol [Akkoorath et al., ICDCS'16]
- GentleRain uses global stable time mechanism [Du et al., SoCC'14]
- Eiger protocol with explicit dependency checks, write-only txns [Lloyd, NSID'13]
- Eventual consistency

Transaction protocols

- Option `clocksi` uses "Cure" protocol to define snapshots and causal dependencies [Akkoorath et al., ICDCS’16]

- Option `gr` uses an implementation of the Gentle Rain protocol to define snapshots and causal dependencies [Du et al., SoCC’14]

Things on our agenda

- Upgrade to Erlang 19
- Partial replication
- Flexible data storage backend
- Security: Access control
- Java client interface
- Feedback welcome!

Sources

- Code repository

  https://github.com/SyncFree/antidote

- Documentation

  http://syncfree.github.io/antidote

- EU-Project Syncfree

  https://syncfree.lip6.fr

People

Deepthi Devaki Akkoorath, Alejandro Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguica, Marc Shapiro, Christopher Meiklejohn, Michał Jabczynski, Santiago Alvarez Colombo, Peter Zeller, Mathias Weber, …. and probably a few more!