Scalability, Confidentiality, Availability, & Integrity For BLOCK-WEBS WITH CHARLOTTE

Isaac Sheff

Hi, I'm Isaac.

And I'm here to talk about …

S.C.A.I.F. BLOCK-WEBS WITH CHARLOTTE

Isaac Sheff
https://IsaacSheff.com
isheff@cs.cornell.edu

Andrew C. Myers andru@cs.cornell.edu

Robbert van Renesse rvr@cs.cornell.edu

“Safe” block-webs with the Charlotte framework.

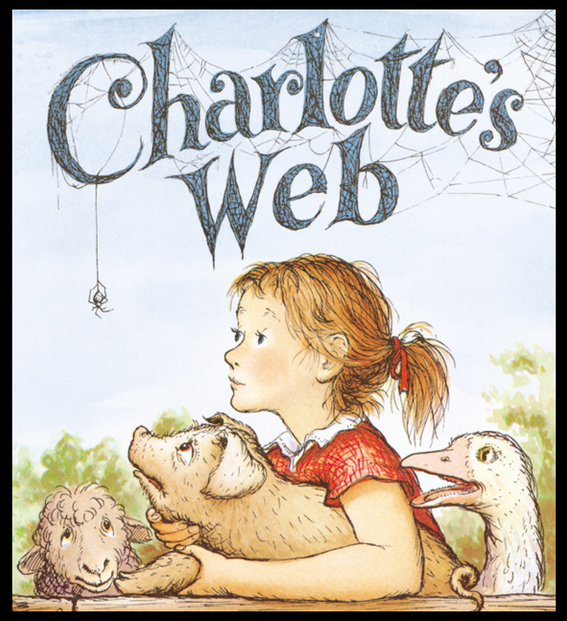
I’m a Ph.D. Candidate at Cornell, working with Andru and Robbert.

I actually signed up for a 5 minute “lighting” talk, looking to get some feedback on Charlotte, but I ended up with this half hour,
so feel free to let me know if you have any comments or suggestions,
this is still very much work in progress.

Also, if you’re the kind of person who likes copious notes, you can find these slides on my website.

S.C.A.I.F. BLOCK-WEBS WITH CHARLOTTE

BLOCKCHAINS

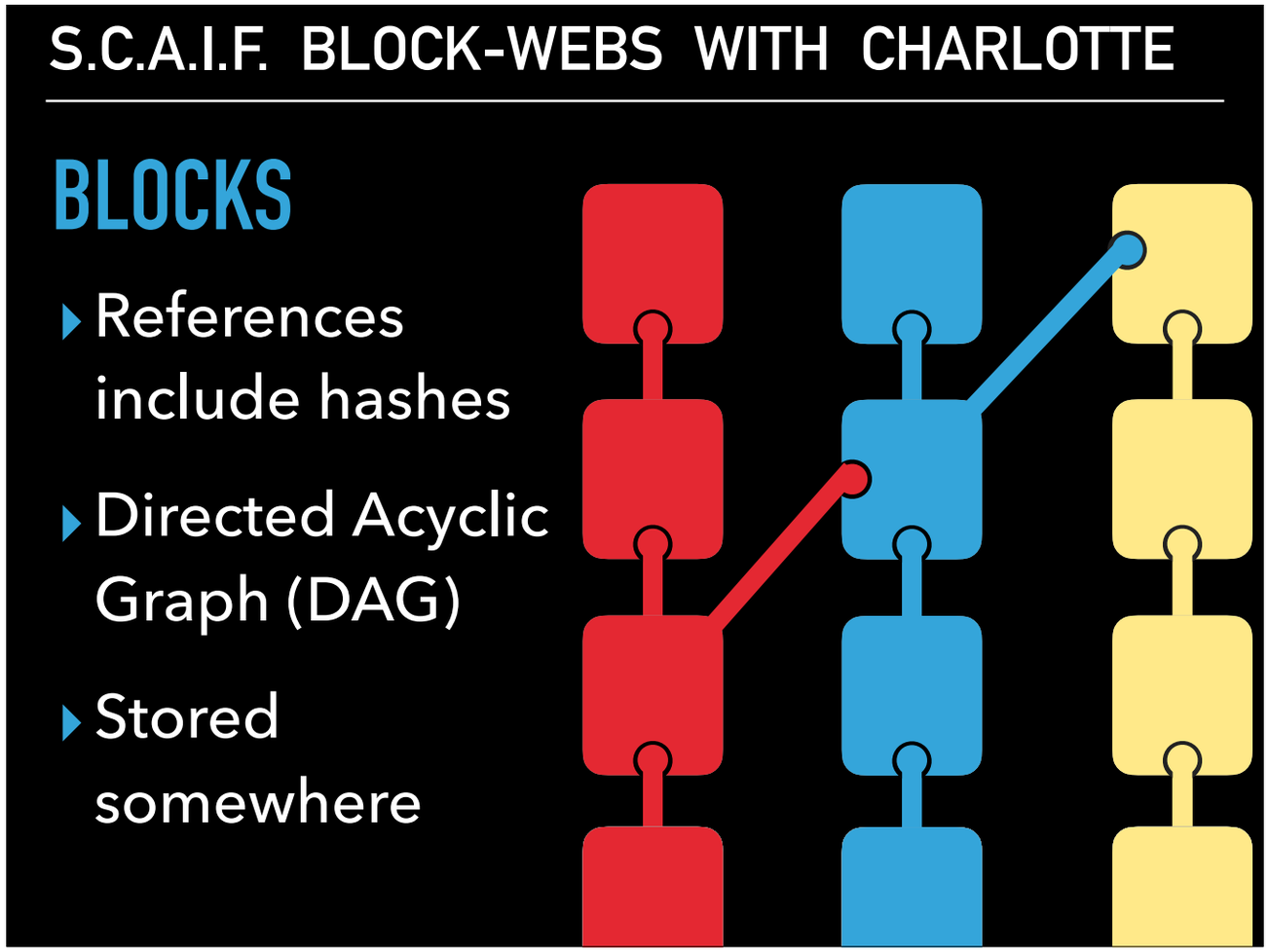


- Data “blocks”
- Self-authentication
- Immutability
- Distributed
- Consensus
  - Proof of work / stake / time / whatever

Let me start with this:
What would we need if we were going to build blockchains anew, from scratch?

Well, fundamentally, data are stored as “blocks."
Many applications seem to want the blocks to stand on their own, they’re “self-authenticating.”
Immutability is often touted as an advantage: past actions can’t be hidden.

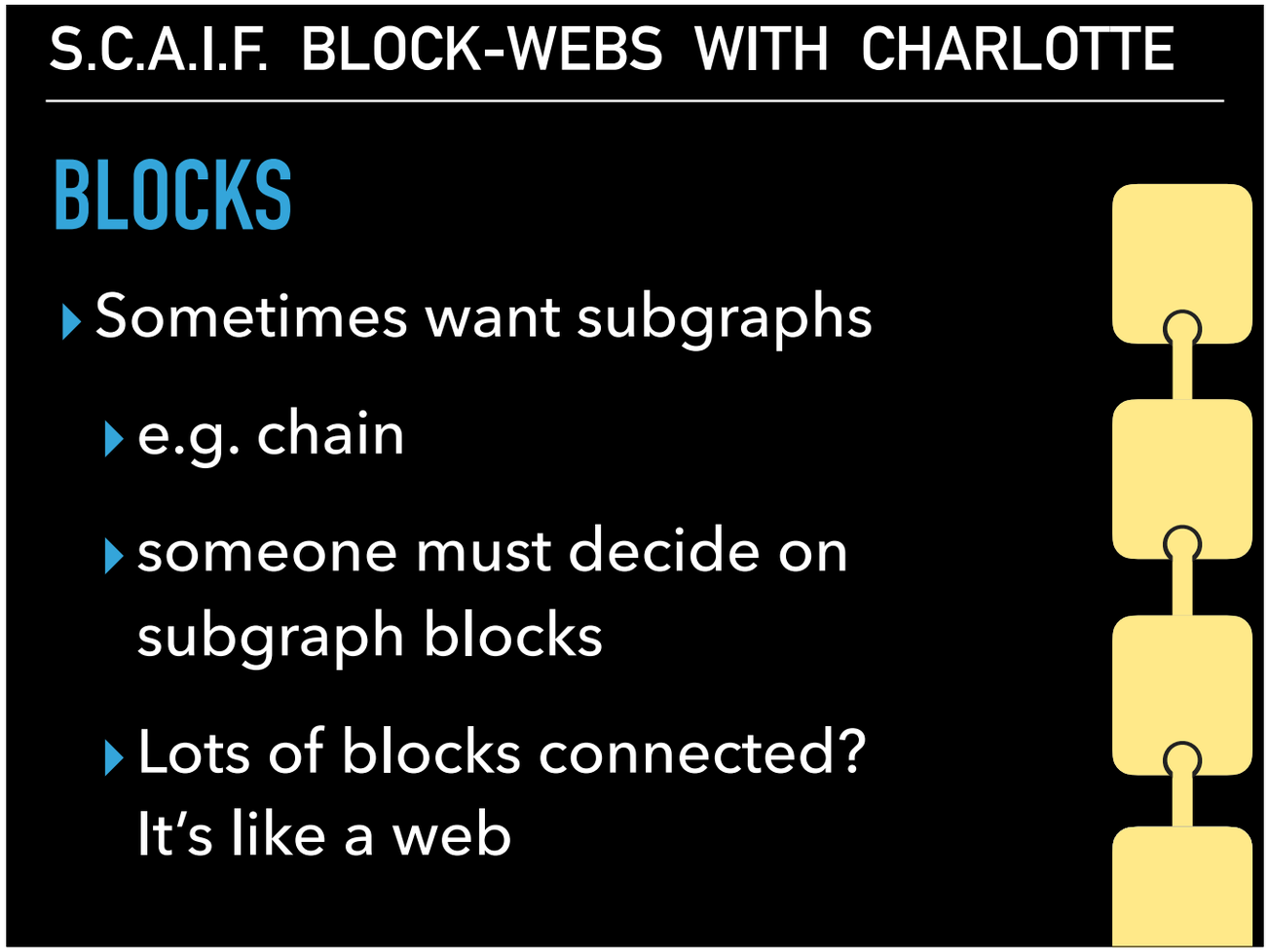
And there’s a lot of talk about what to use to decide which blocks “belong,” in the event of 2 equally valid blocks, like double spends.



But at the most basic level, blocks are just a unit of data, which reference each other by hash.
The only thing this really implies about the structure of blocks is that they're a directed acyclic graph, a DAG.

Of course, each has to be stored somewhere, so this is in some sense a distributed DAG.

Now, for most applications, not just any old DAG will do.
We want to select some subset of the DAG of all blocks, say, these yellow ones …



And thereby get some more useful properties.

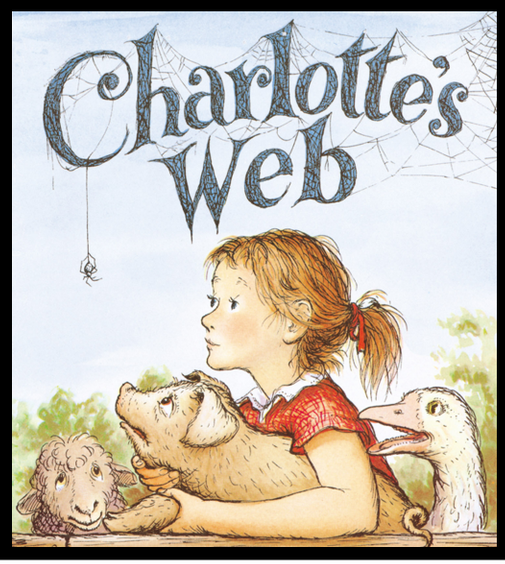
For instance, we might select a chain.

Ultimately, someone has to pick out, for each data structure one might care about, which blocks belong and which don’t.
We can call each of these sub-DAGs a block-web, for lack of a better term.

S.C.A.I.F. BLOCK-WEBS WITH CHARLOTTE

KEY IDEA

- Separate Storage from Consensus



So given just these very general notions of what block-webs should be, we set out to make a framework,
wherein everyone could make whatever kind of blocks they like, whatever kind of storage and consensus they like, with a uniform format for at least block parsing, references, that kind of thing.

As it deals with spinning webs, I'm tentatively calling it charlotte, after the children's book.

And our key insight, which we think might be helpful, is to separate storage from consensus.
In all the block systems I know of, basically everyone has to store all the blocks, but there's no particular reason that need be the case.

S.C.A.I.F. BLOCK-WEBS WITH CHARLOTTE

KEY IDEA

- Separate Storage from Consensus
- **WILBUR** Availability Servers
  - Purpose is to stay alive

So we'll call the set of computers tasked with storing stuff "Wilbur" Servers,
named for the character in the book who's only purpose is to stay alive.

Each Wilbur server will store blocks for whatever purpose it chooses.
However, servers can provide proof that they're storing specific blocks, and attestations that they'll continue doing so.

S.C.A.I.F. BLOCK-WEBS WITH CHARLOTTE

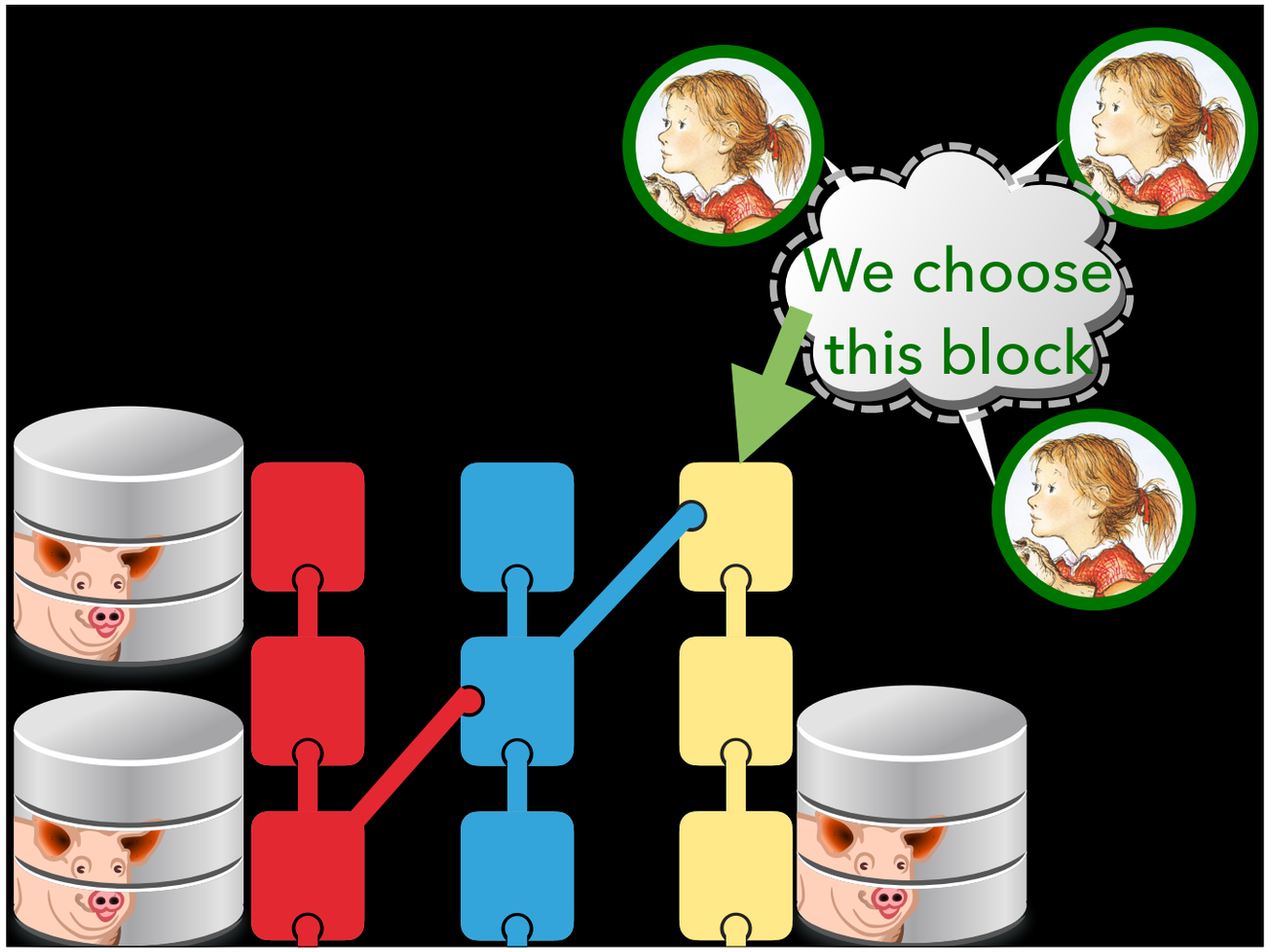
KEY IDEA

- Separate Storage from Consensus
- **WILBUR** Availability Servers
  - Purpose is to stay alive
- **FERN** Integrity Servers
  - Make hard choices

And we'll call the servers who select which blocks belong in a given Block-Web "Fern" servers,
after the character from the book who ultimately decides who lives and who dies.

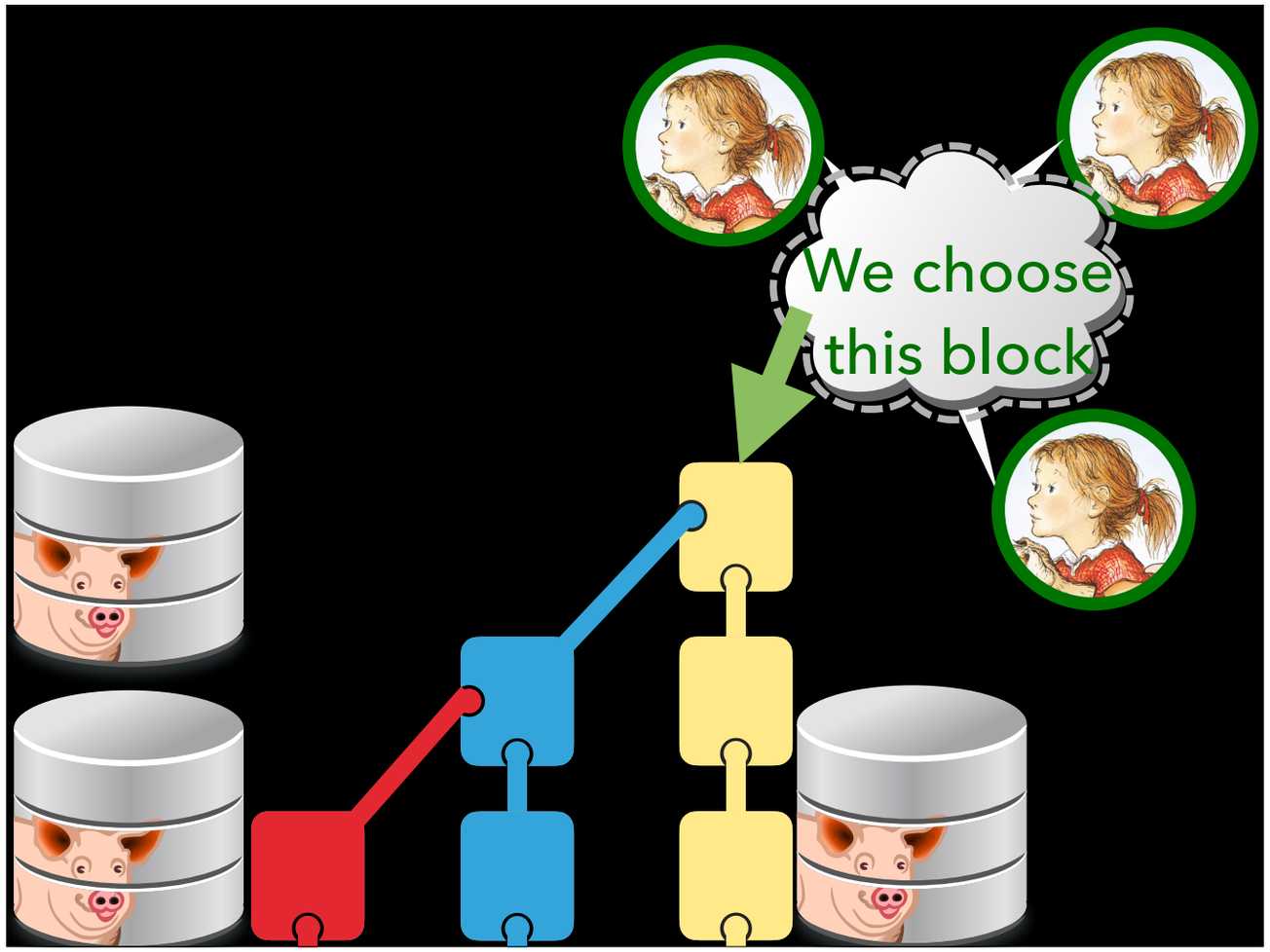
These servers provide some kind of attestation, such as a proof of work, or record of a consensus algorithm,
which indicates that a block belongs in a given data structure.

Fern servers' motives and selection criteria are their own:
perhaps they have standards on where a block must be stored before they'll consider approving it.



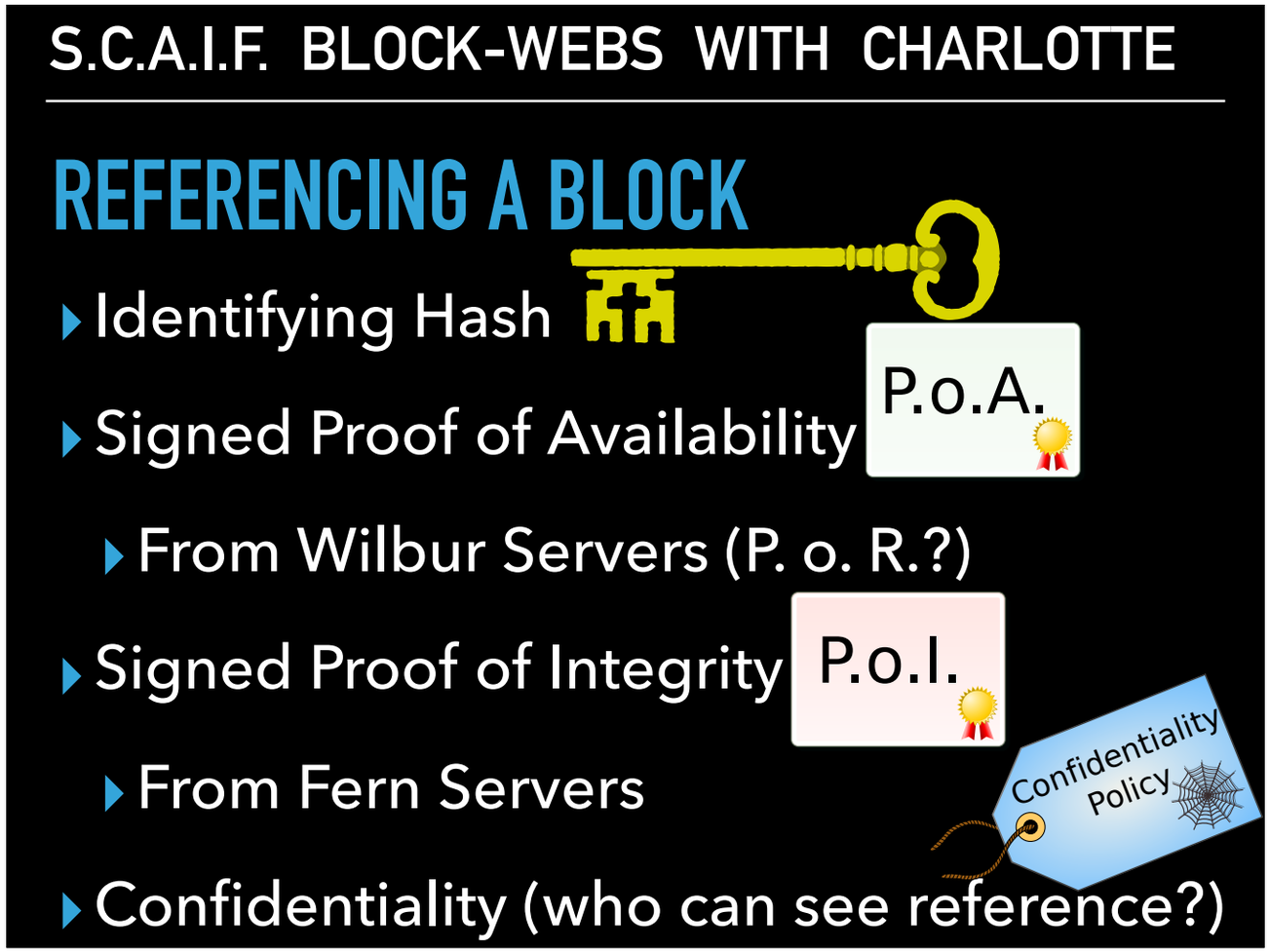
So in general, there will be some DAG of blocks, stored across the various Wilbur Servers, and
if you're interested in reading a specific data structure,
you should start with a block selected by the Fern servers relevant to that web.

Naturally, that block will reference others, so you may end up wanting to …



read its DAG ancestors.

Blocks will have to specify in their references which ancestor belongs in which data structure, but all of that is application-specific.



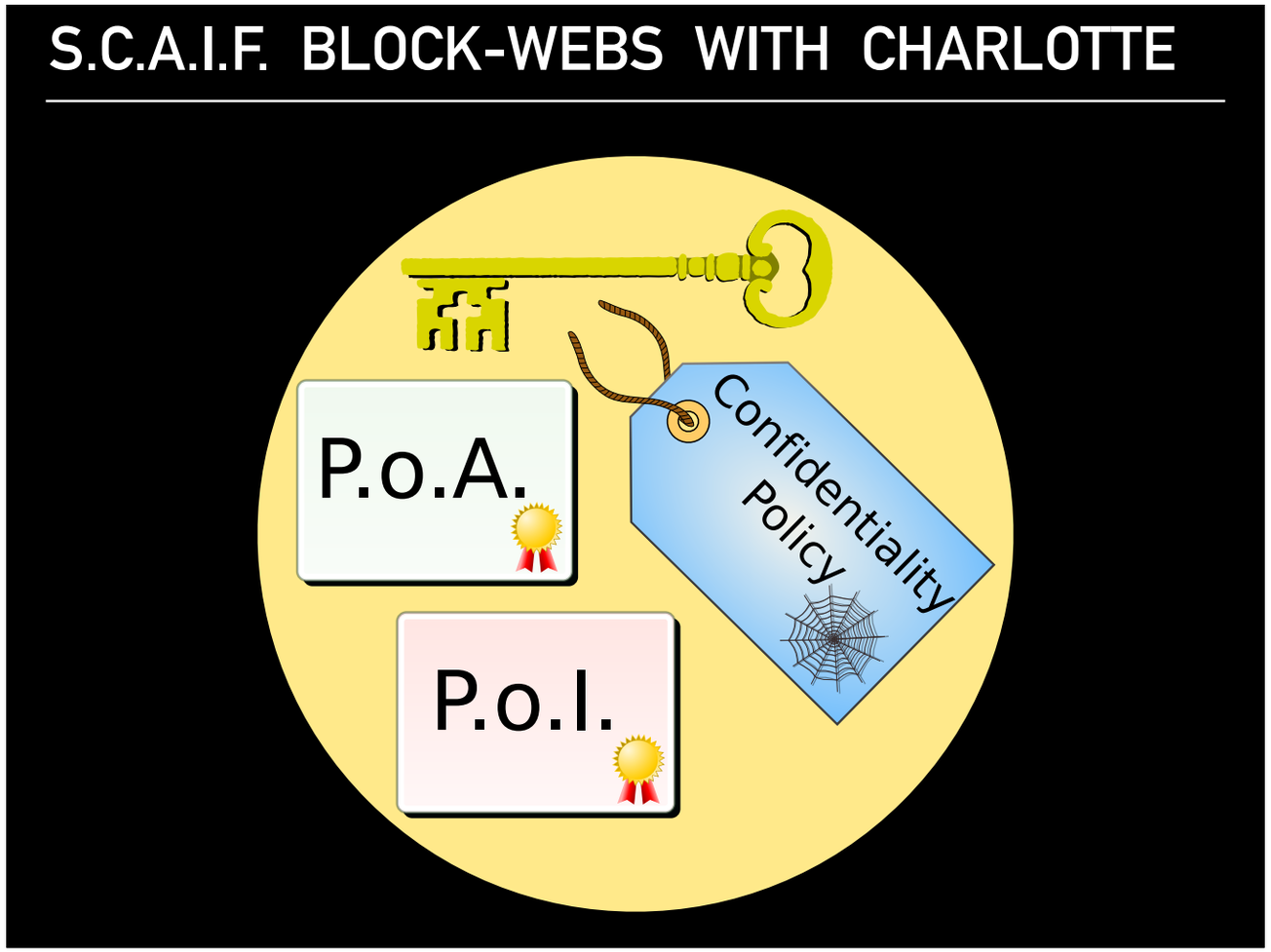
So ultimately, when you want to reference a block, you might want 4 things:

There's the block hash, which forms some kind of lookup key,
and lets you verify when you indeed have the correct block.

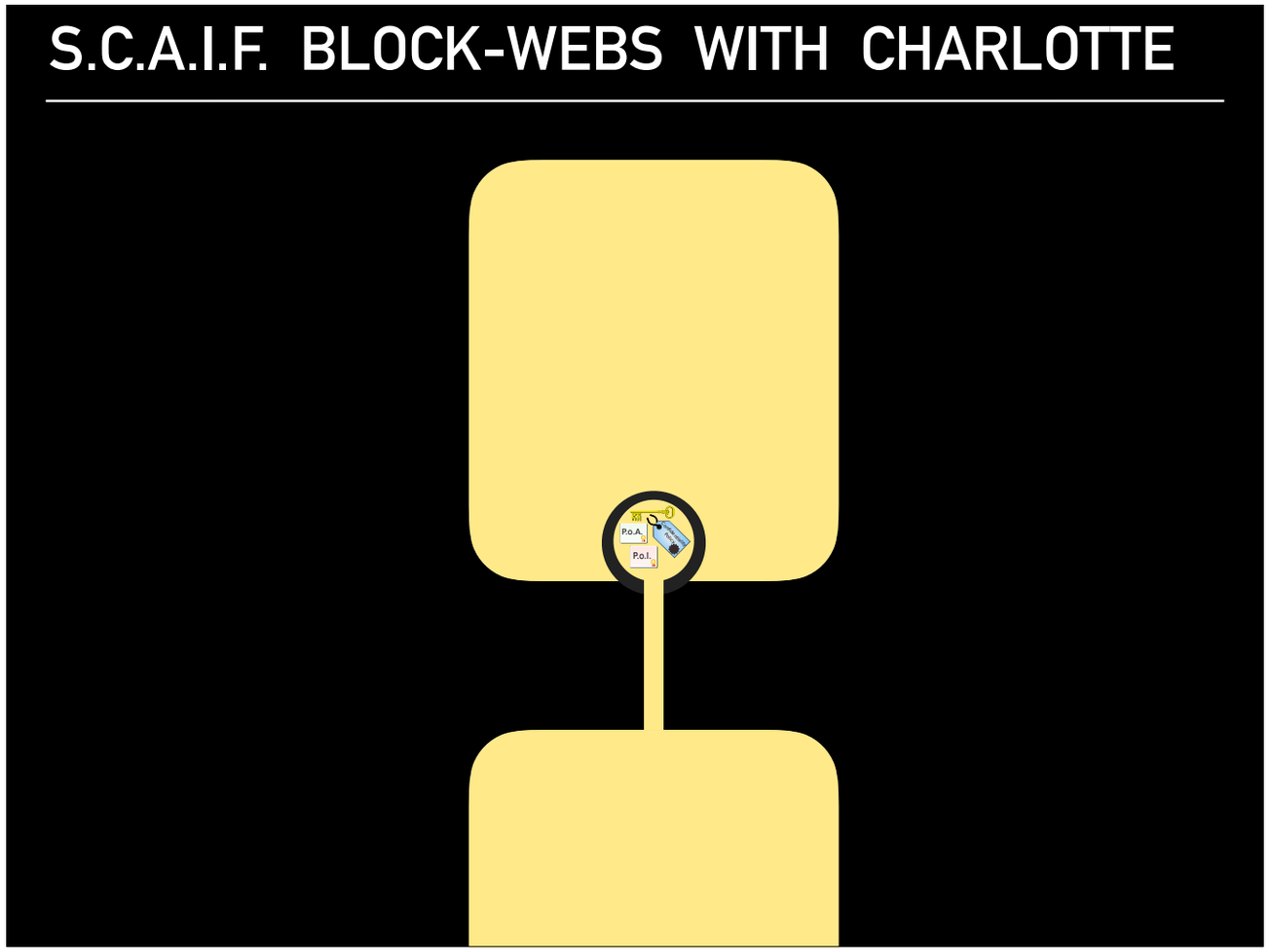
There's a proof of availability, constructed from the attestations of Wilbur Servers,
which tells you where you can expect the block to be stored.

There's a proof of integrity, constructed by the relevant Fern servers,
which tells you which Webs this block belongs in,

And there may be a confidentiality policy, specifying to whom this reference may be passed.
I'll get to a little more on that later.

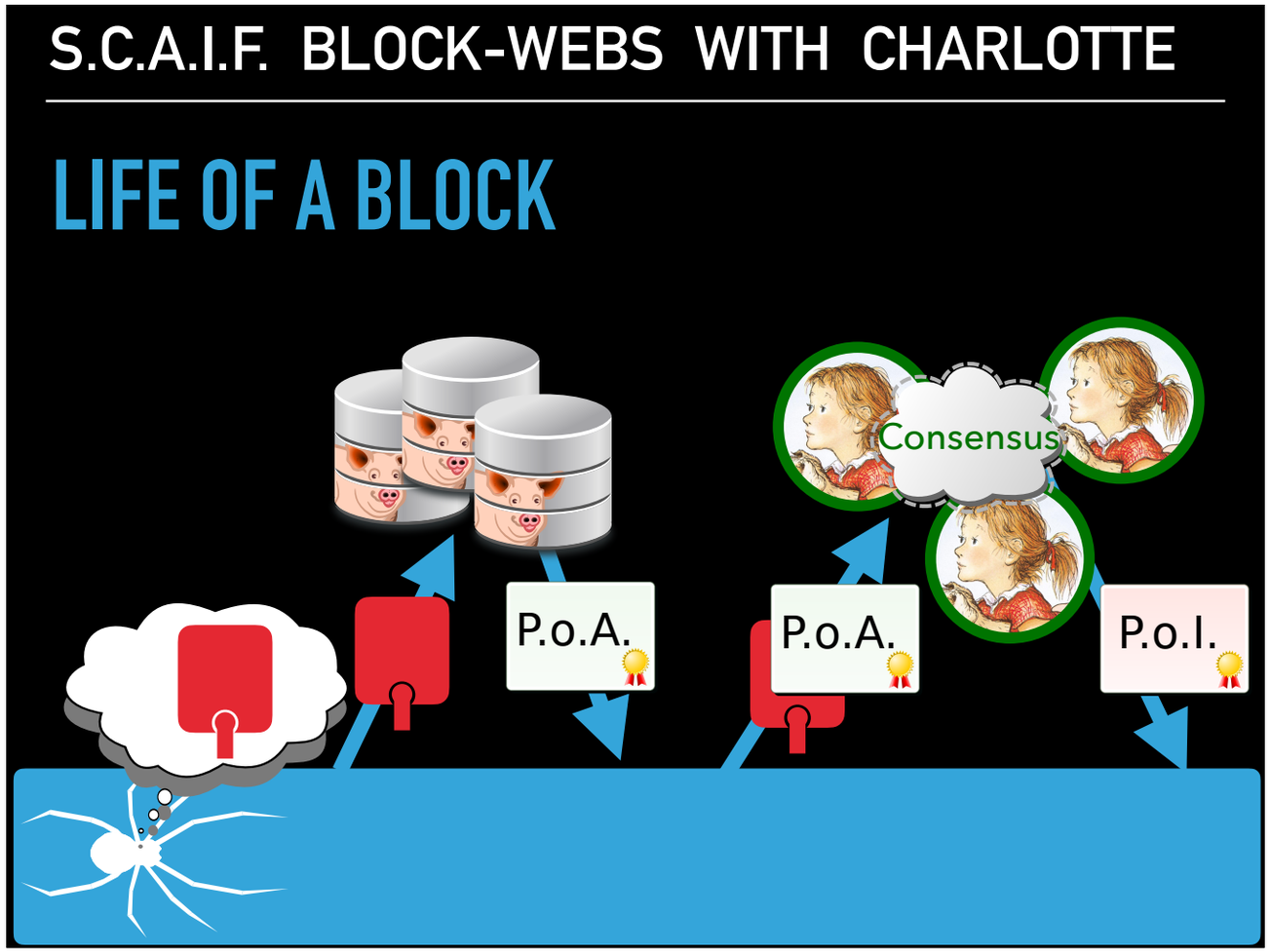


All these things together form a block reference, and



each time one block refers to another, it should carry such a reference

So that anyone retrieving the block knows this information.



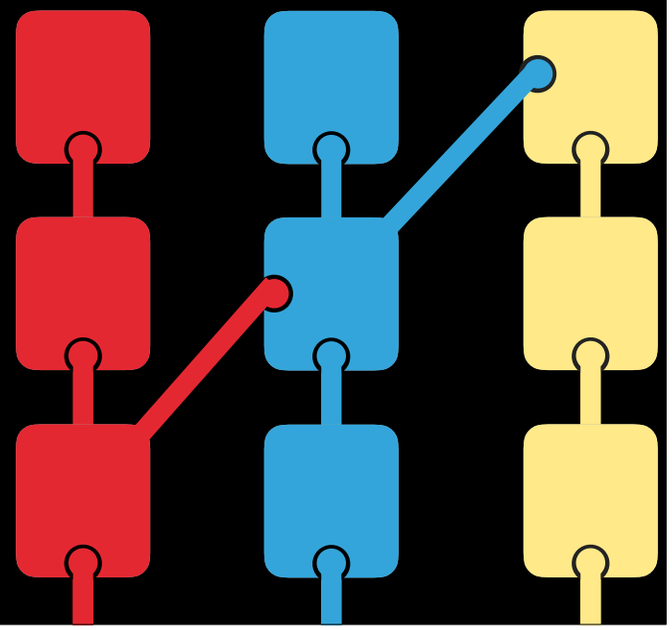
So if our spider friend here wants to mint a block, she creates the block, and
then stores it on some Wilbur servers to get a proof of availability,
then passes it to some relevant Fern servers, and
asks them to approve it for membership in a given data structure,
to obtain a proof of integrity.

From this she can construct a reference to the block.

S.C.A.I.F. BLOCK-WEBS WITH CHARLOTTE

SCALABILITY

- No total order
- Independent blocks
  - in independent data structures
  - on independent servers



This general mindset lends itself to a lot more scalability than traditional block-chains.
We don’t impose any unnecessary total order on everything, although Fern servers might for some Webs.
We allow independent data structures to follow their own rules and requirements on independent servers, while enabling them to reference each other in a standard way.

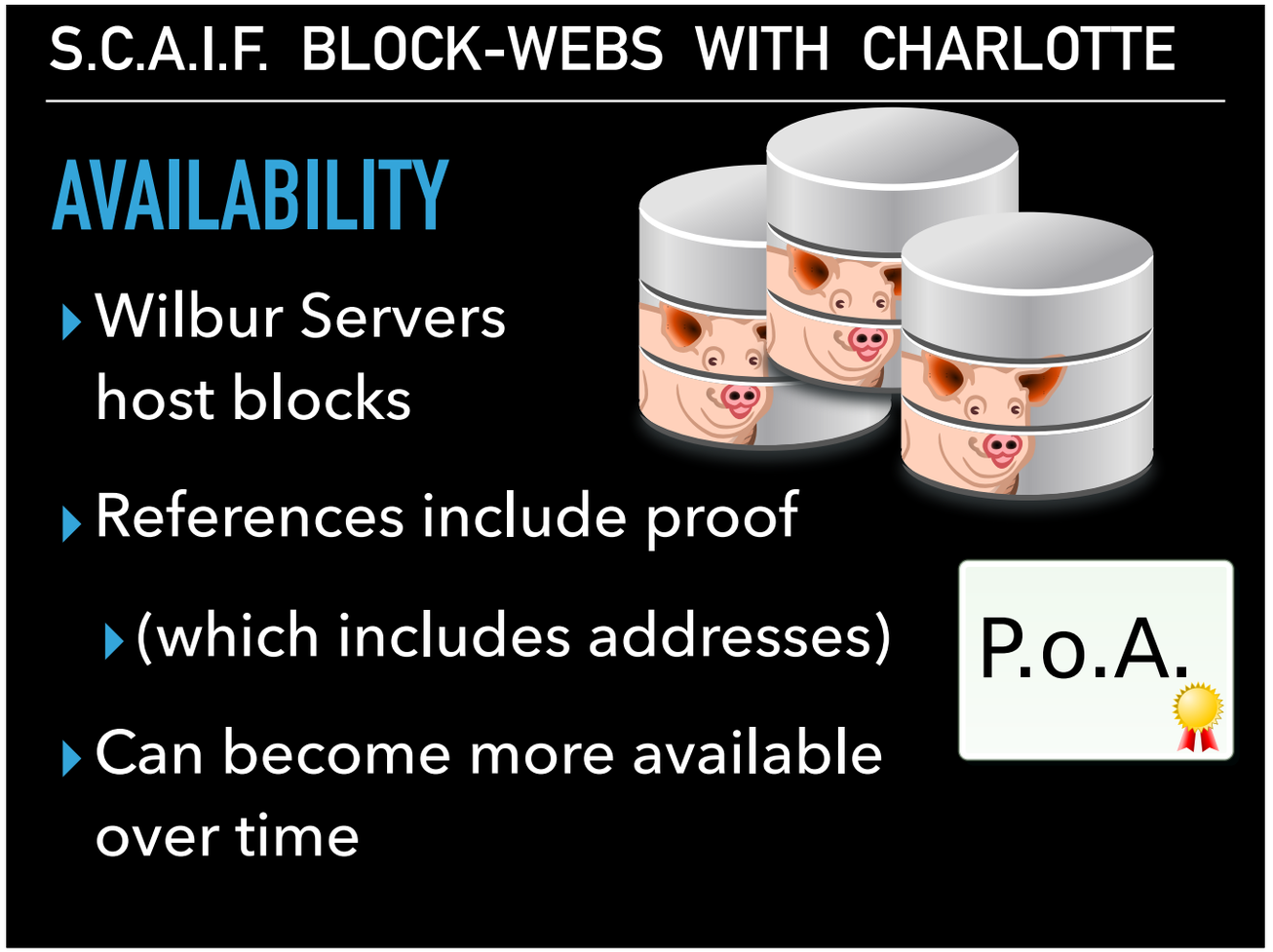
S.C.A.I.F. BLOCK-WEBS WITH CHARLOTTE

CONFIDENTIALITY

- Encrypt blocks
- For each reference in a block:
  - Block encryption must satisfy reference's confidentiality policy

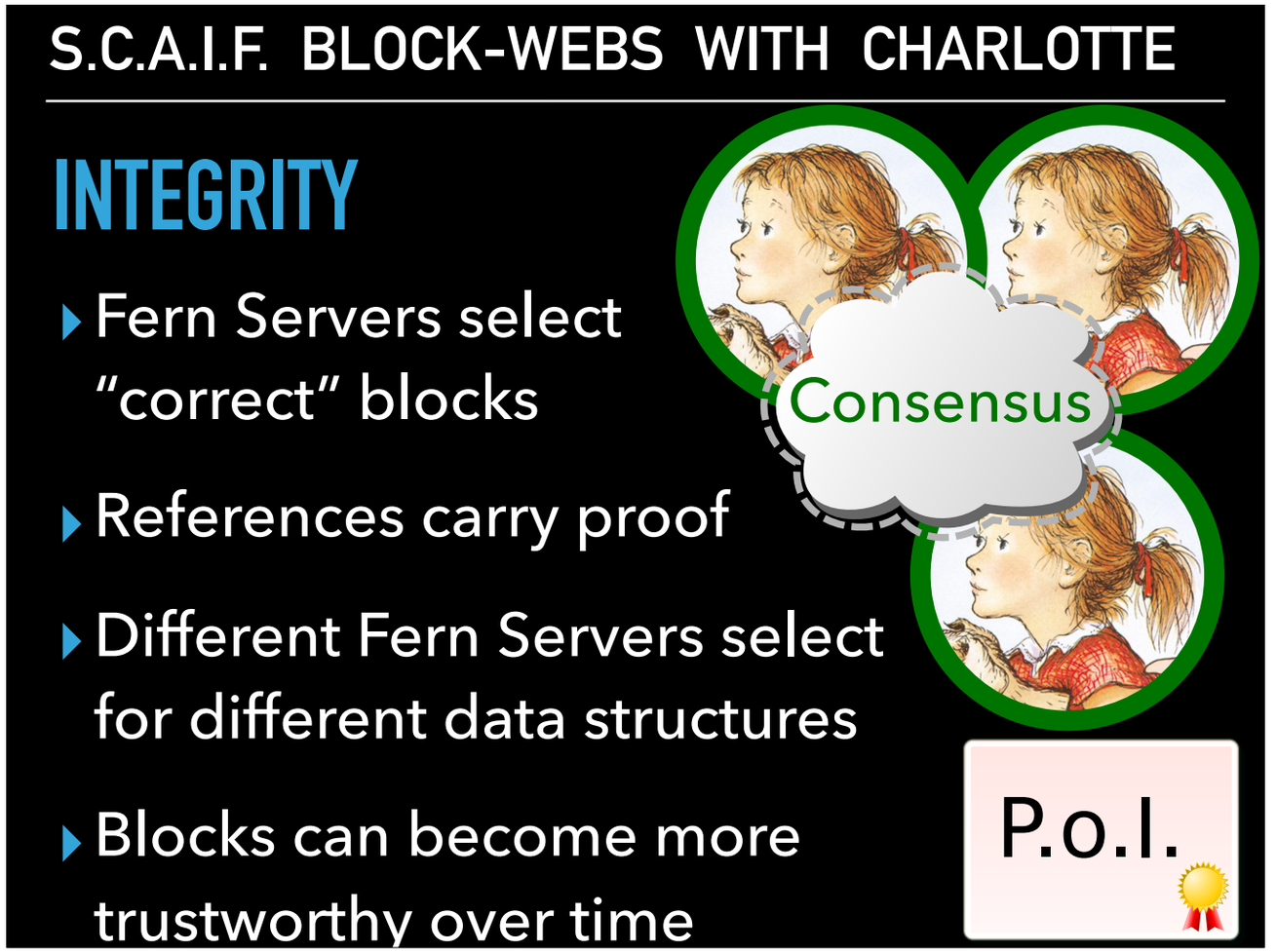
When it comes to confidentiality, we can of course encrypt block contents, but we can do a little better.
Sometimes, the existence of a block is a secret.
So each block should contain a confidentiality policy, specifying who can know it exists.
Each reference contains a copy of that policy, and a proof (if we make our blocks merkle trees), that that is the block's policy.
Any block with that reference should be encrypted such that only people allowed to see the reference can read it.
Any Fern or Wilbur server who signs off on a block that does otherwise, has signed a proof that they're untrustworthy.

And that really is the best we can do.
After all, anyone who knows a thing can ultimately tell others out-of-band about that thing.



Wilbur servers make blocks available, and each Web can set its own requirements for the Wilbur attestations necessary for a block to join. They might require it be stored at specific places, or perhaps in sufficiently many places.

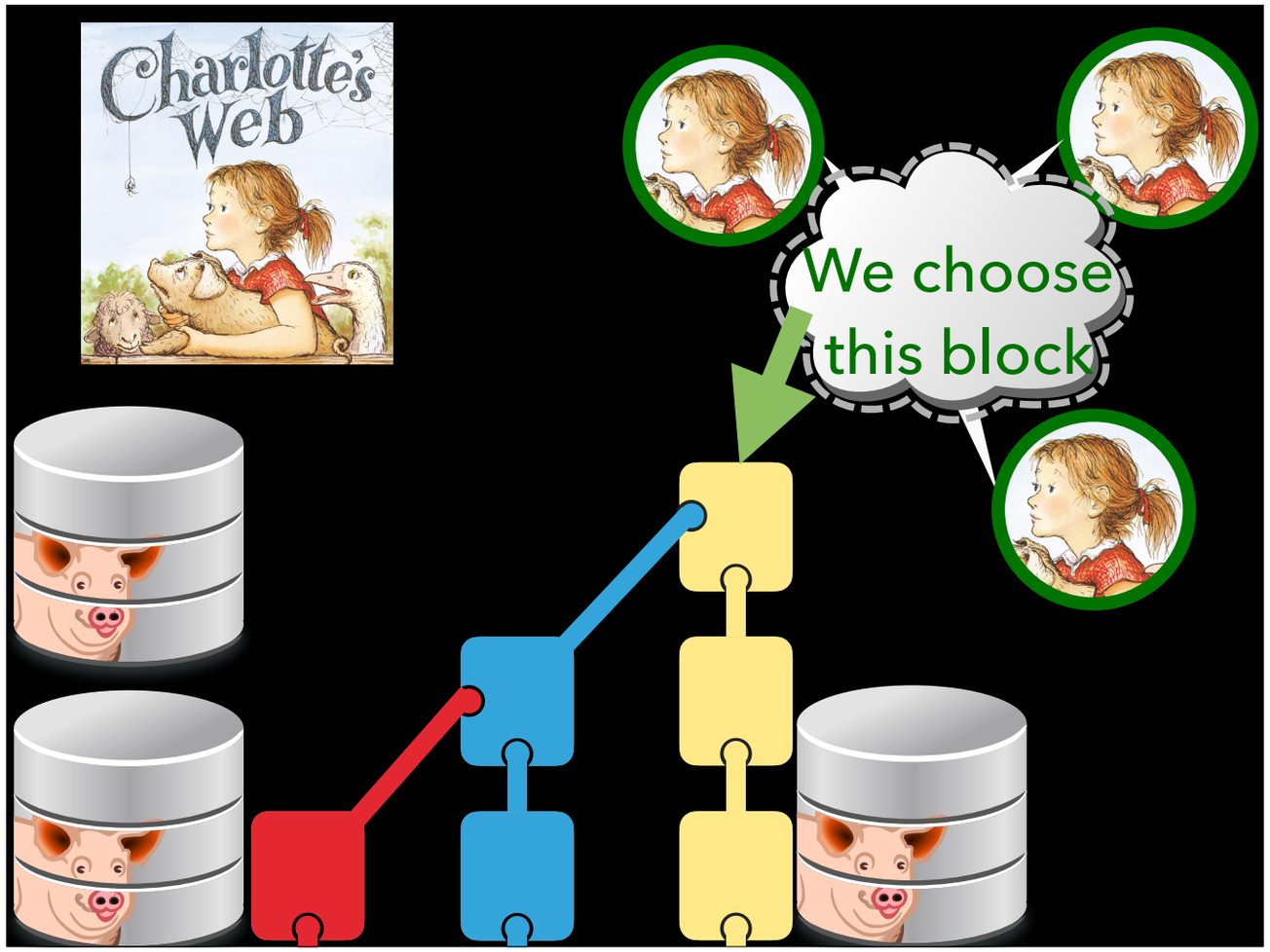
As time goes on, more servers may store the block, so future references can construct stronger proofs of availability.



Likewise, Fern servers lend integrity to each Web.
Different Webs may use different consensus mechanisms and different servers.

ultimately, however, each fern server lends its integrity to the statement "this block belongs in this Web."
Over time, more and more fern servers may sign on to that statement, solidifying a block's membership in future references.
There are other ways, too, that a proof of integrity can be bolstered in future references:

For instance, each time a block is added to a proof of work chain, we can construct a higher integrity proof for all older blocks in the chain.



So that’s the core idea of Charlotte, a framework for future block-webs.
When it’s ready, hopefully, you all will want to build all your new and exciting block-web projects in Charlotte.

Now’s not a bad time for questions before I go on.

PAUSE

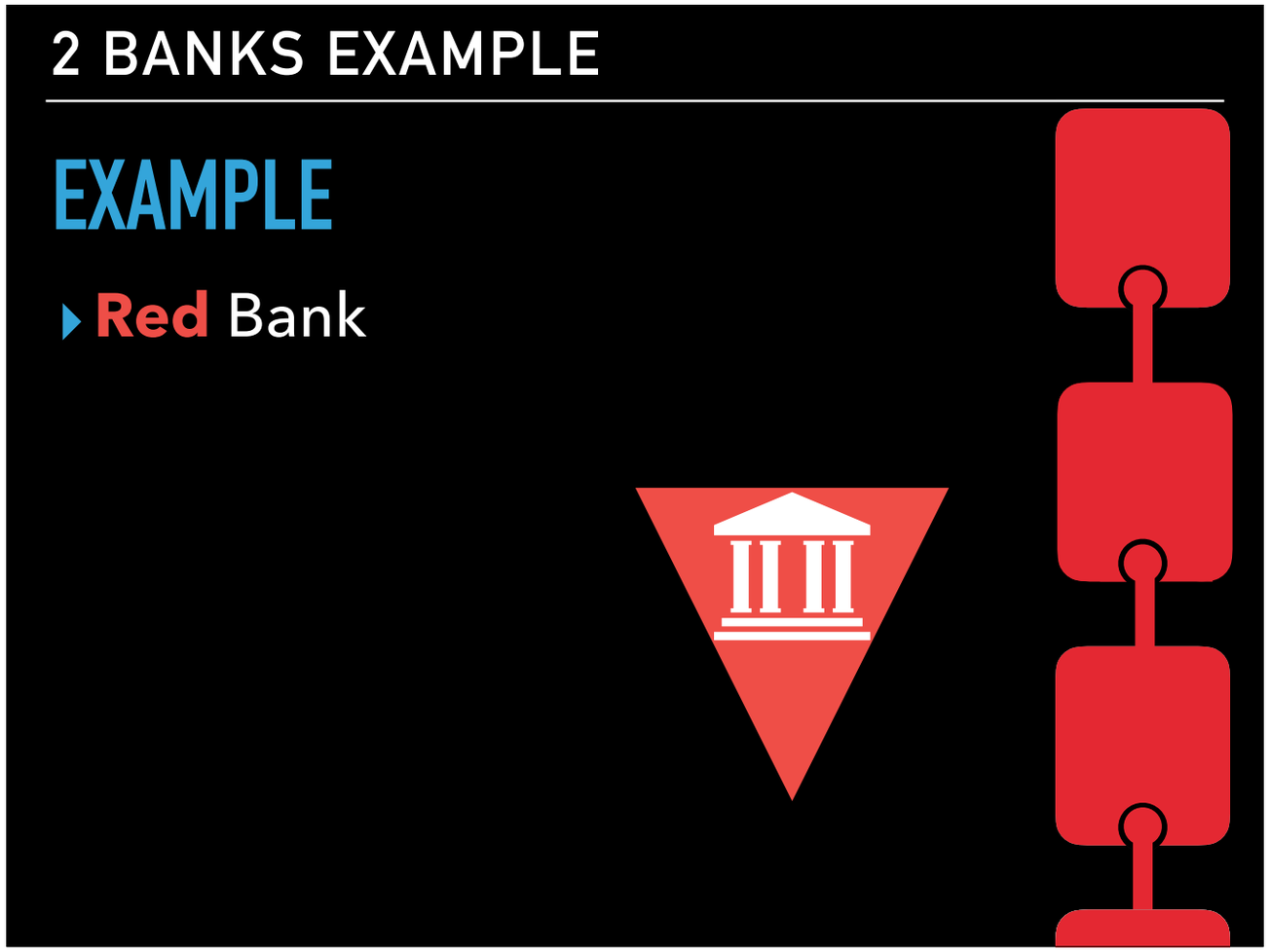
Like I said, it’s still very much a work in progress, but if you have any ideas, I’d love to hear them.

2 BANKS EXAMPLE

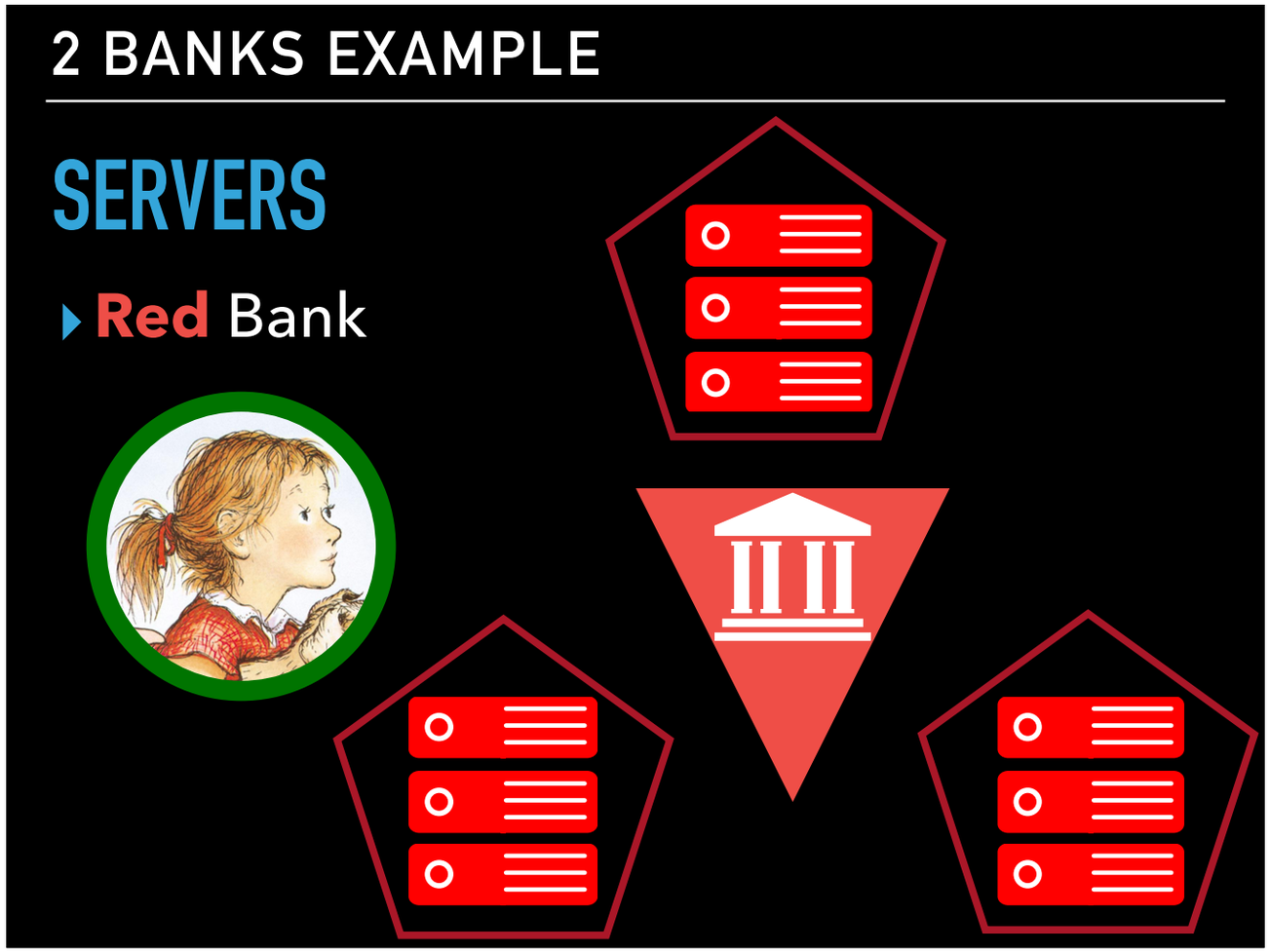
CHARLOTTE USE CASE EXAMPLE

- **Red** Bank

Now I’d like to bring up an example use case for Charlotte.
Suppose a bank, which I’ll call Red, want’s to keep its transactions in a block-chain.



Yes, there are reasons it might want to do something else, but just suppose…



As Red is a private organization, we'll suppose it has a bunch of servers on which it's running its permissioned blockchain.

For the moment, I'm most interested in their Fern Servers: these are the ones deciding whether each new block belongs on Red's block-chain.

And for now I'll just suppose they're divided into groups, maybe they're data centers, who knows, the point is that each contains multiple servers.

2 BANKS EXAMPLE

SERVERS

- **Red** Bank
- 3 groups of servers
- No group goes entirely byzantine
- Simultaneous failures in ≤ 1 group

What's important about these groups are Red's failure assumptions:

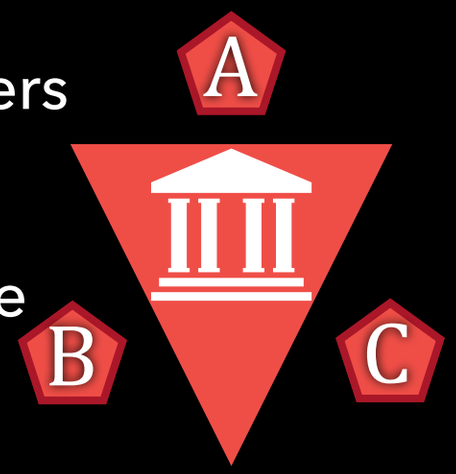
Red assumes that no group will be entirely byzantine:
That is, no group can be entirely hacked by the adversary.

It also assumes that there are not simultaneous failures in more than one group: two groups are functioning completely correctly, even if one completely crashes.

2 BANKS EXAMPLE

SERVERS

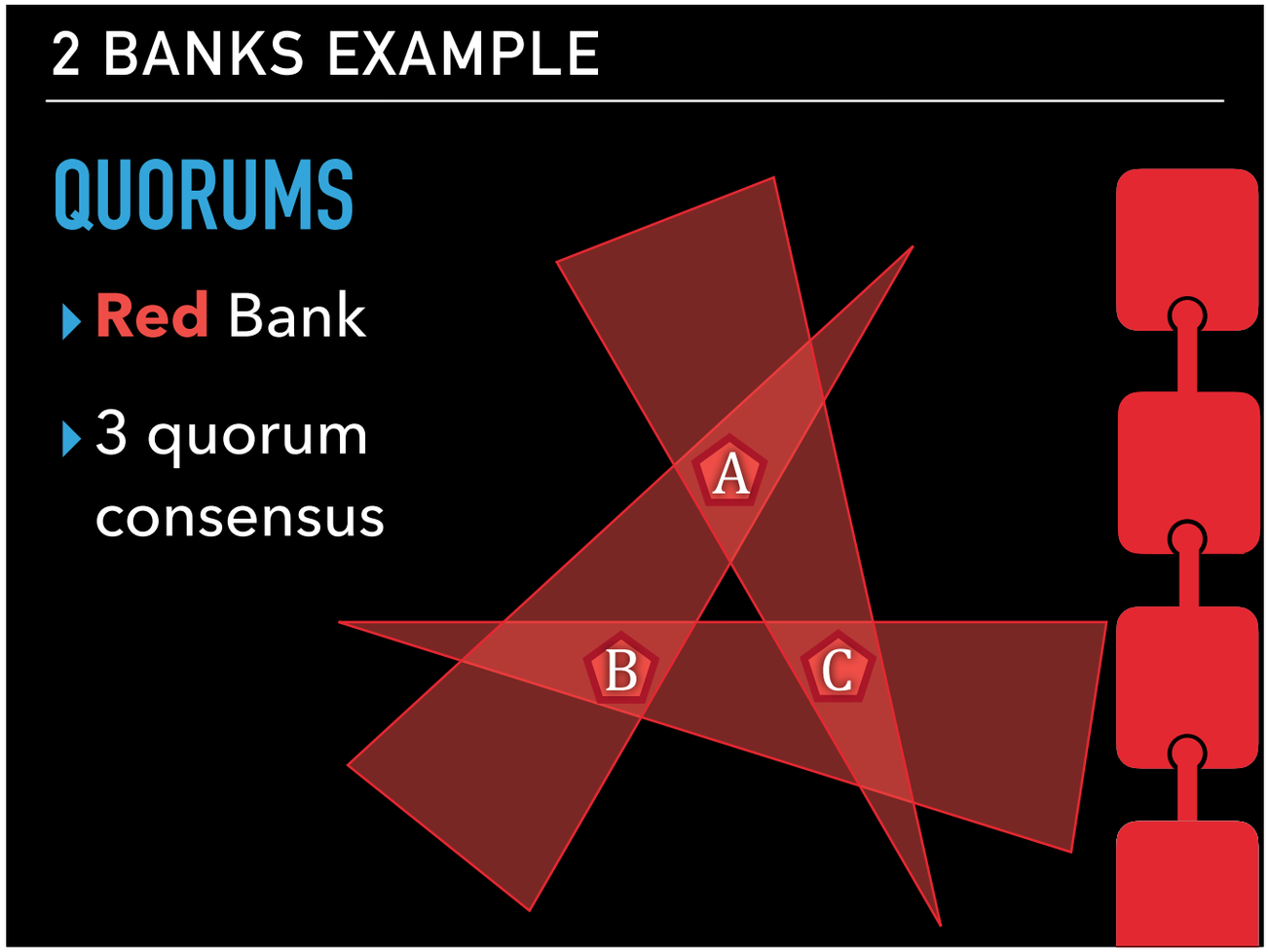
- **Red** Bank
- 3 groups of servers
- No group goes entirely byzantine
- Simultaneous failures in ≤ 1 group



What's important about these groups are Red's failure assumptions:

Red assumes that no group will be entirely byzantine:
That is, no group can be entirely hacked by the adversary.

It also assumes that there are not simultaneous failures in more than one group: two groups are functioning completely correctly, even if one completely crashes.

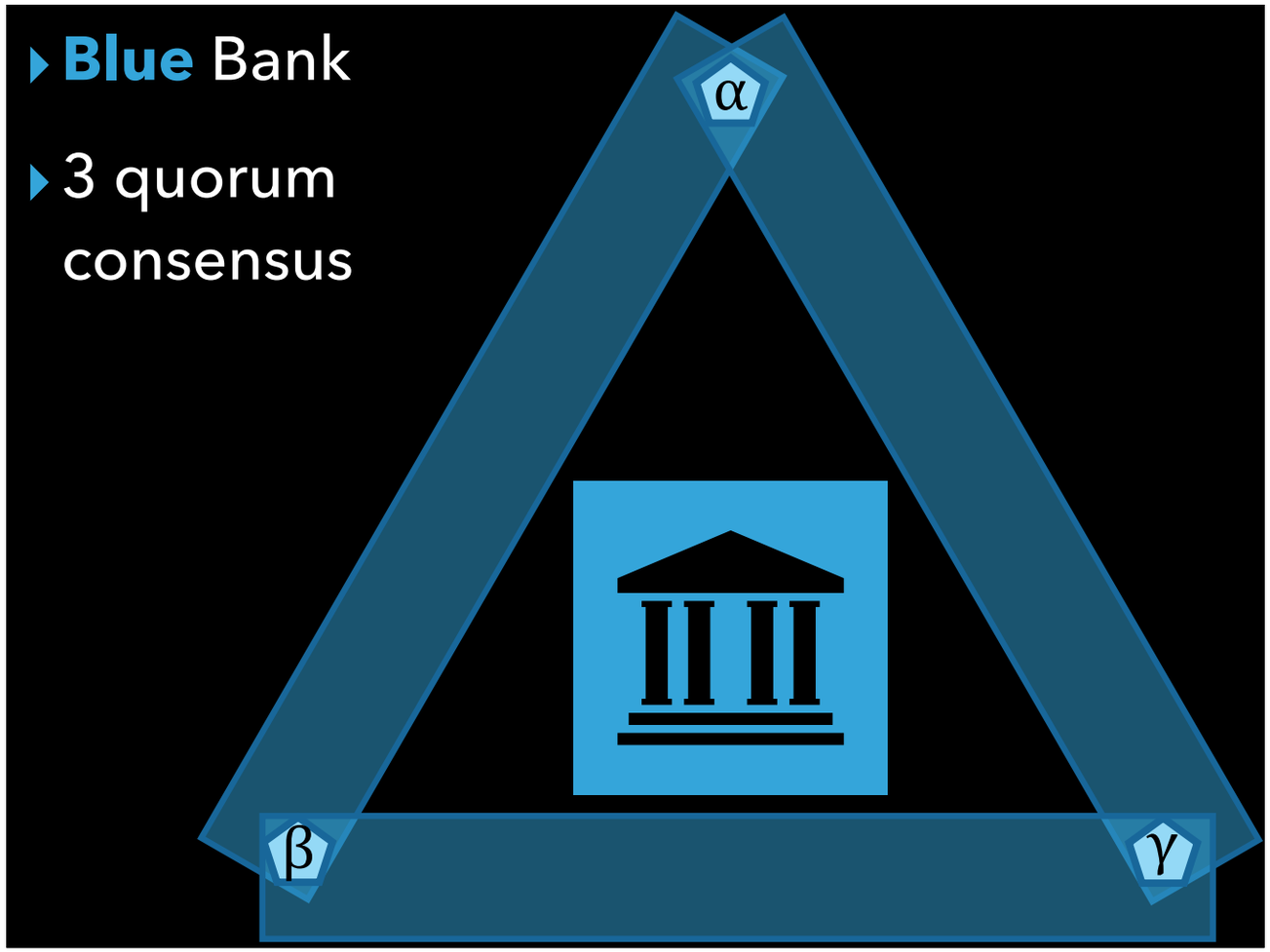


So with these assumptions, Red can run a consensus to approve each block of its chain using these quorums: any two server groups form a quorum.

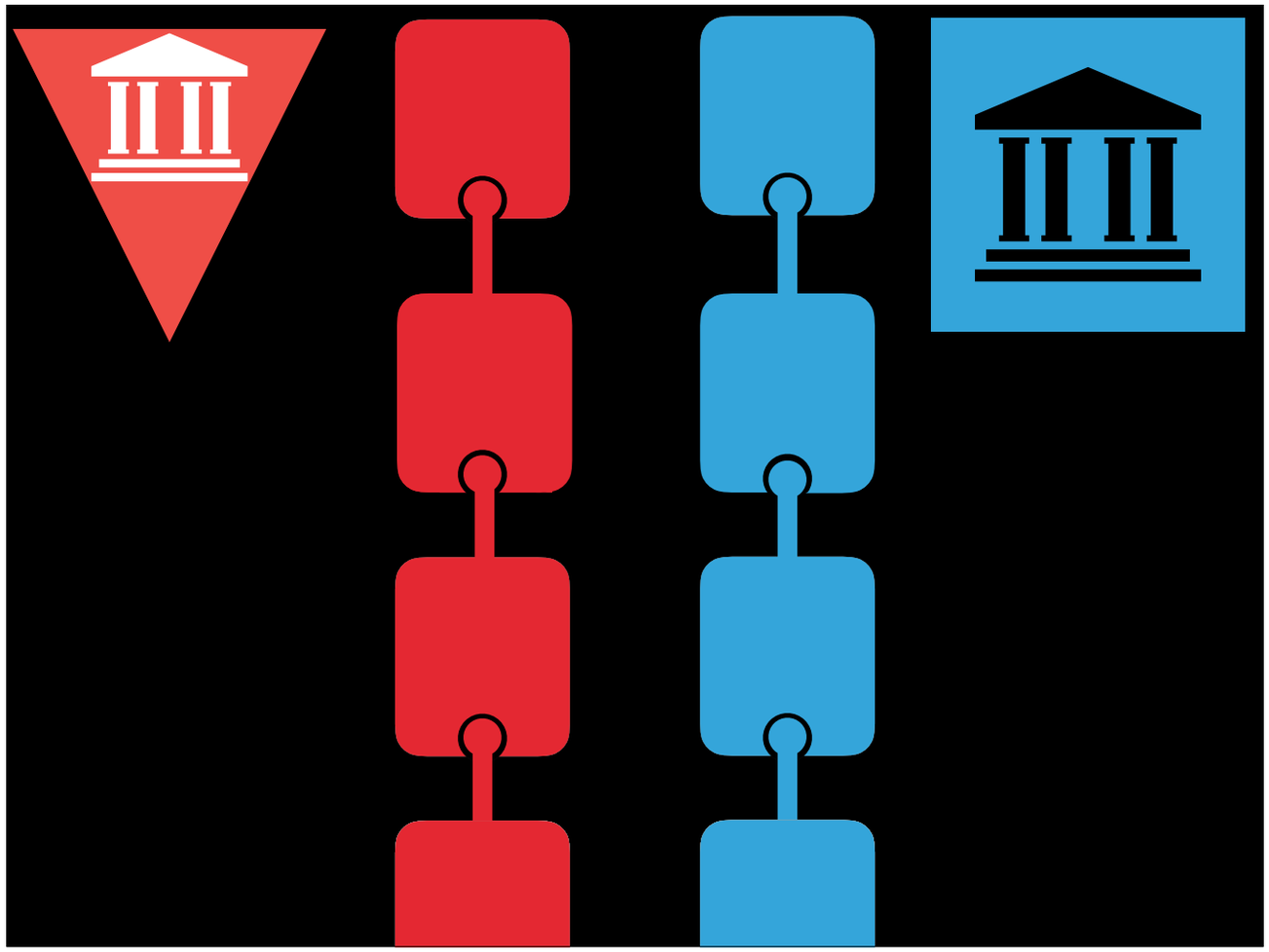
For those of you unfamiliar with traditional consensus algorithms, conceptually, all the servers in a quorum agree together to commit on a value, and that will make a decision irrevocable.

- **Blue** Bank
- 3 groups of servers
- No group goes entirely byzantine
- Simultaneous failures in ≤ 1 group

Suppose another Bank, Blue, has a similar setup…



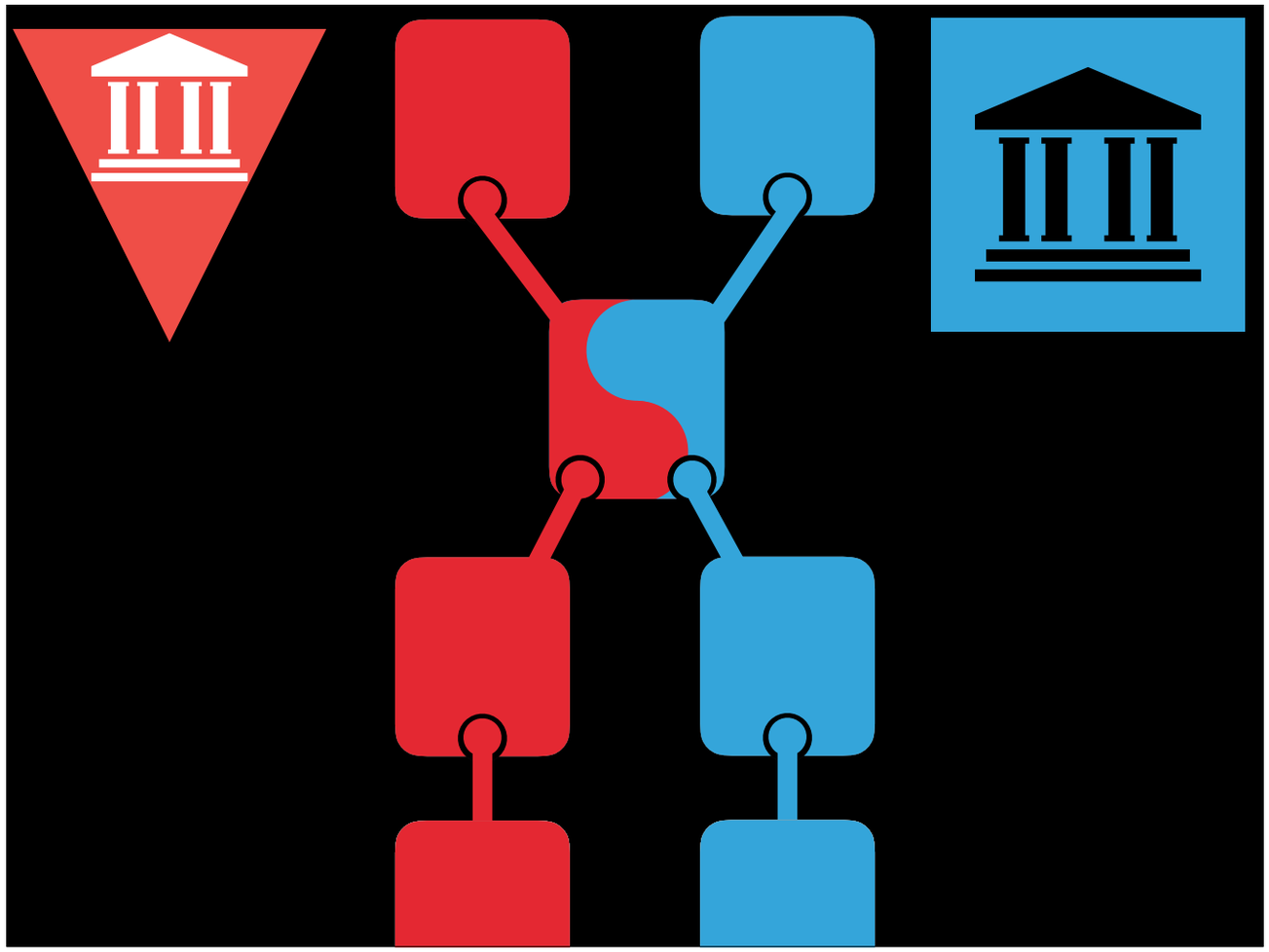
so these would be Blue’s quorums.



So now we have Red and Blue each maintaining a blockchain with their own Fern servers running their own consensuses.

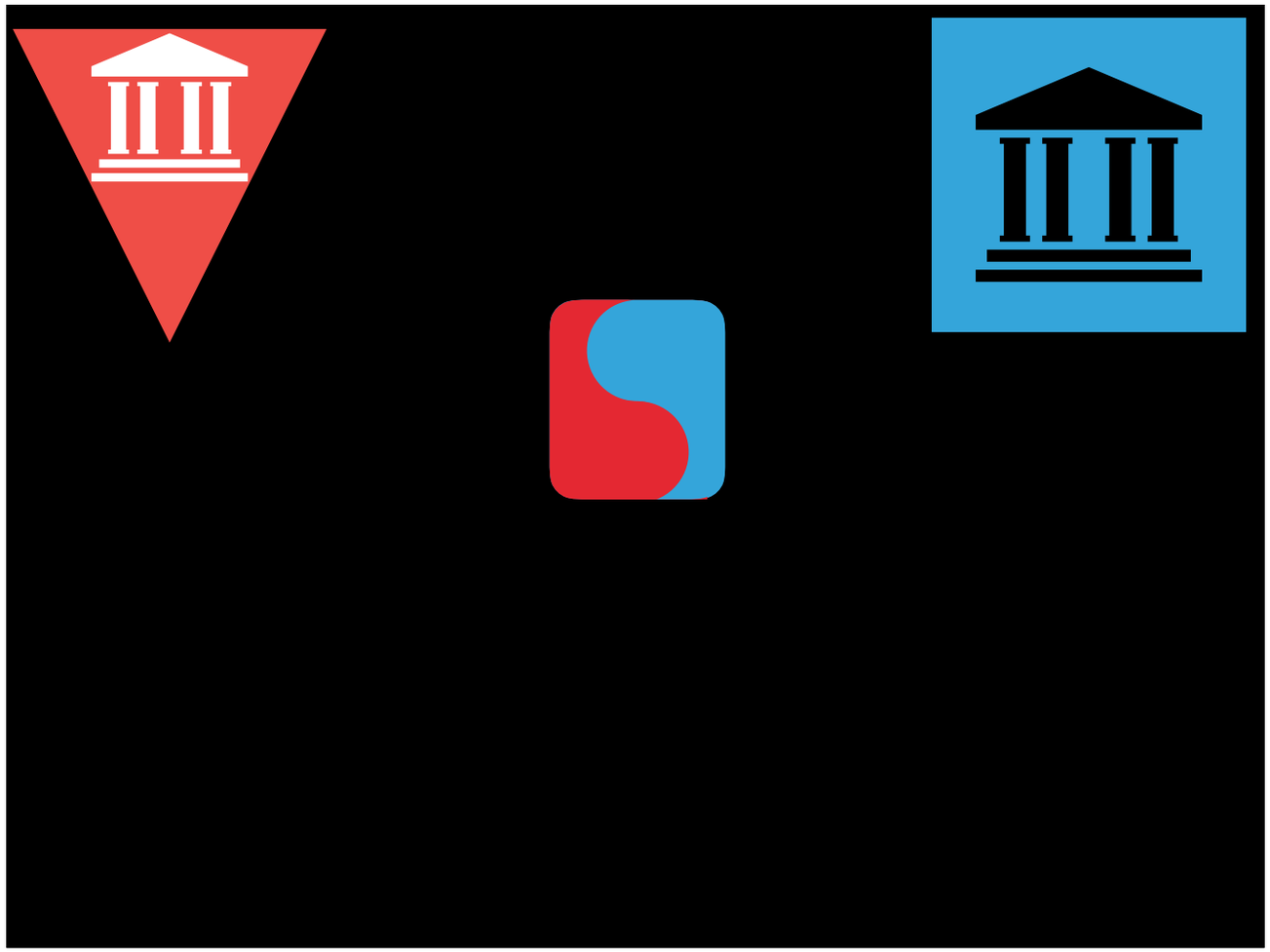
But suppose Red and Blue want to share some transactions.
Perhaps an account at Red Bank wants to send money to an account at Blue bank.

There are a number of inter-ledger commit protocols for this kind of thing, but what we're ultimately simulating here…

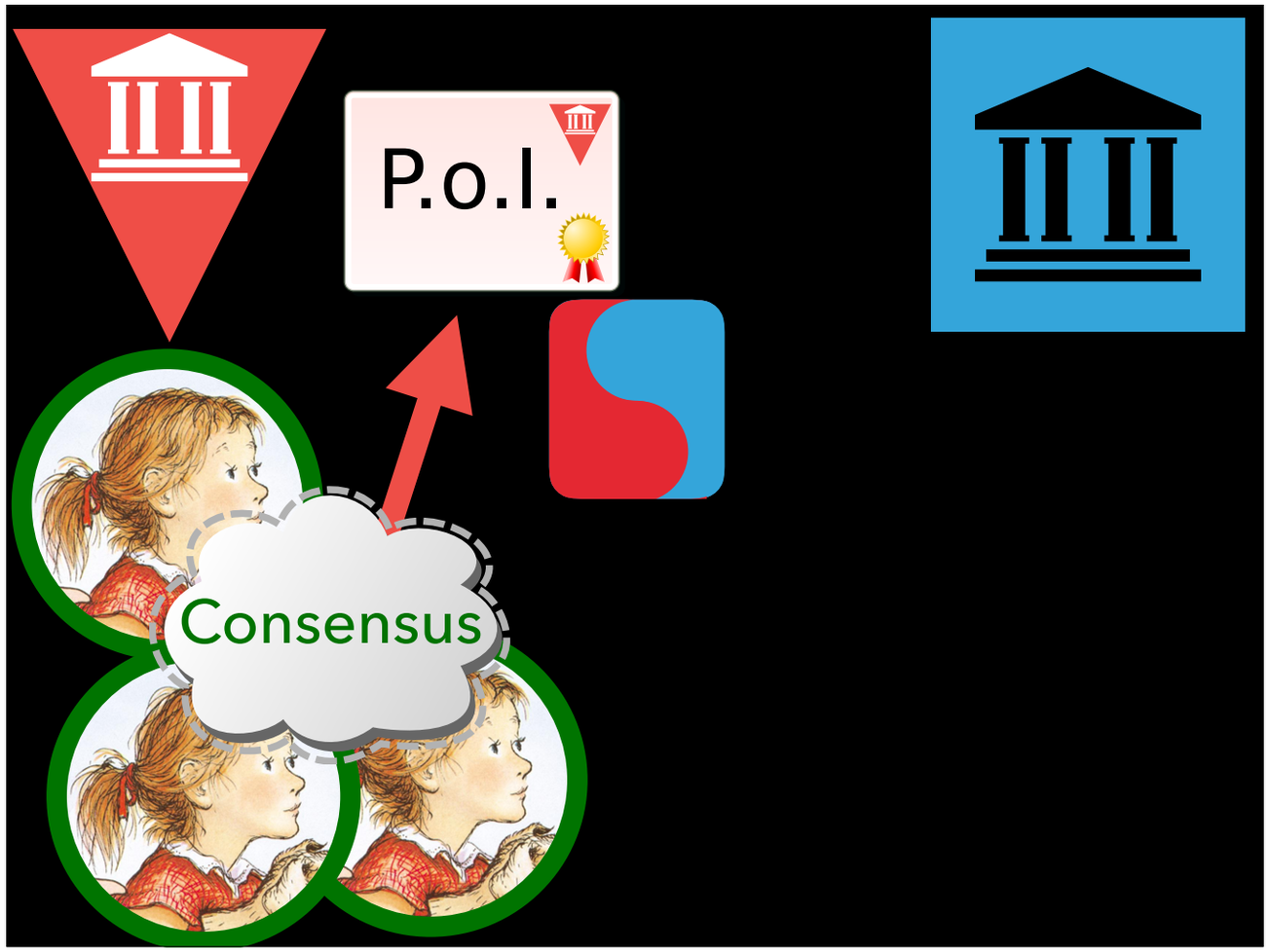


is having at least one block of transactions be in both chains.

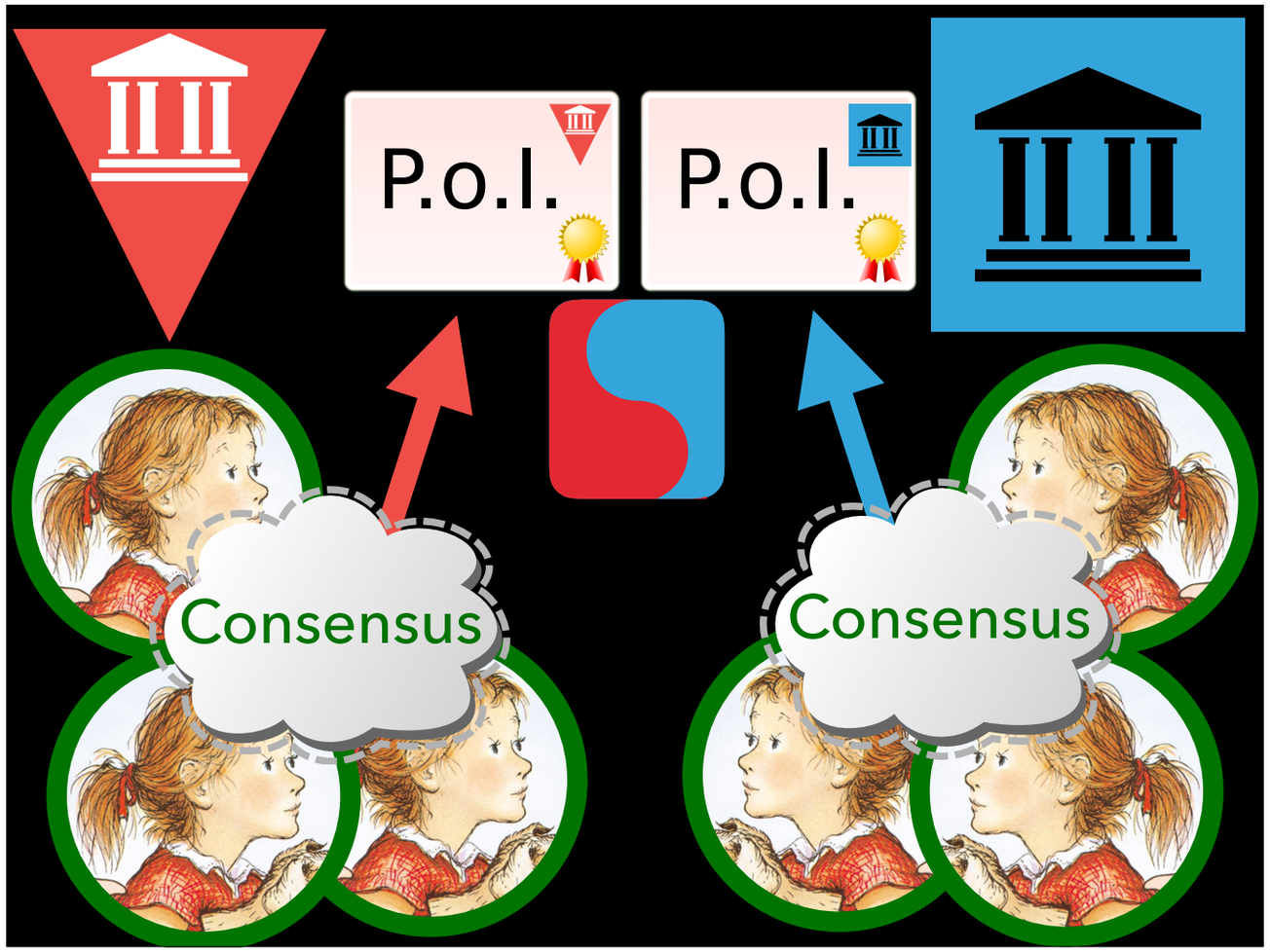
and that is exactly the kind of thing we can do in Charlotte.



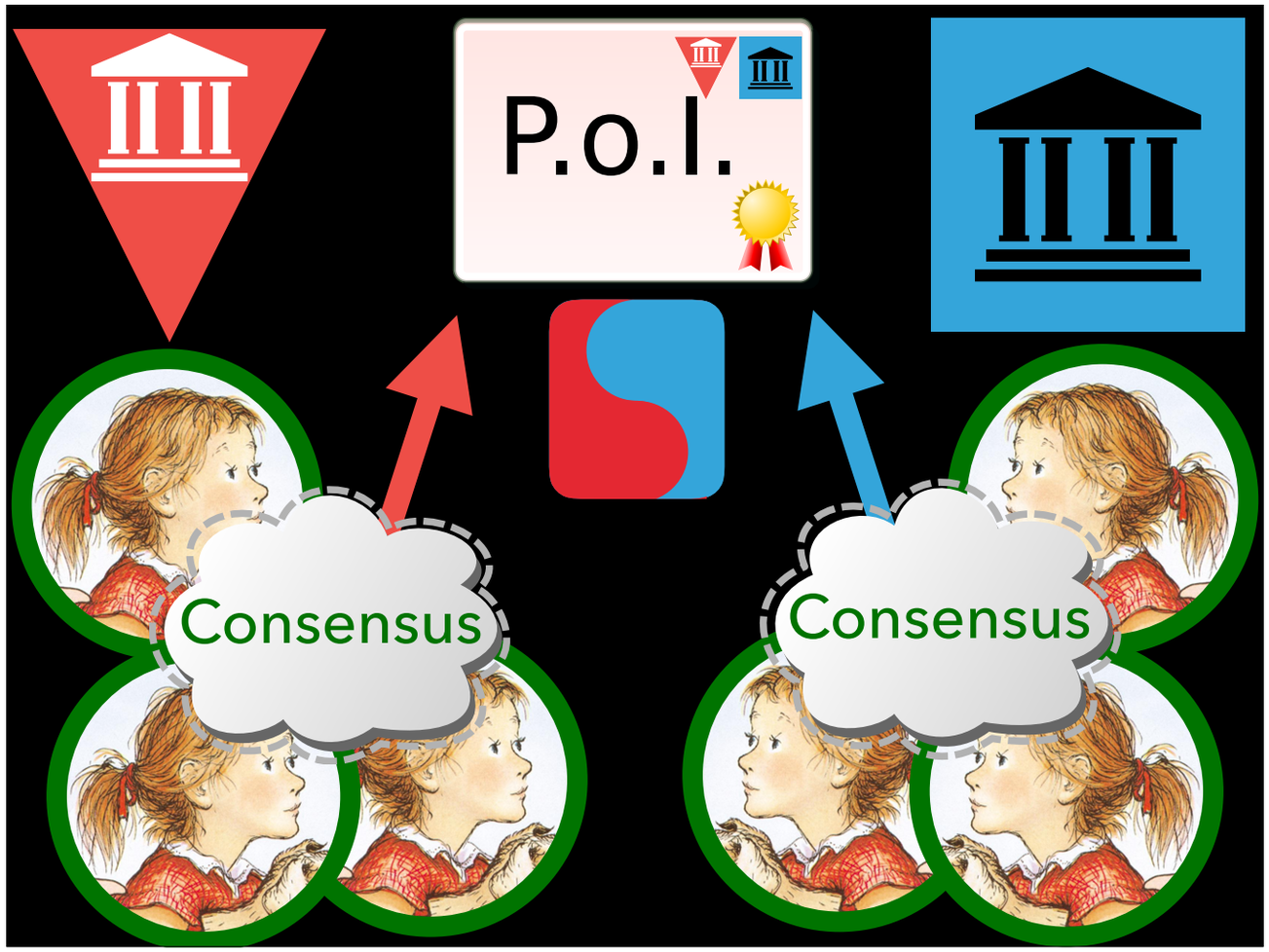
So at least naively, to approve this block, someone would have to create and store it, …



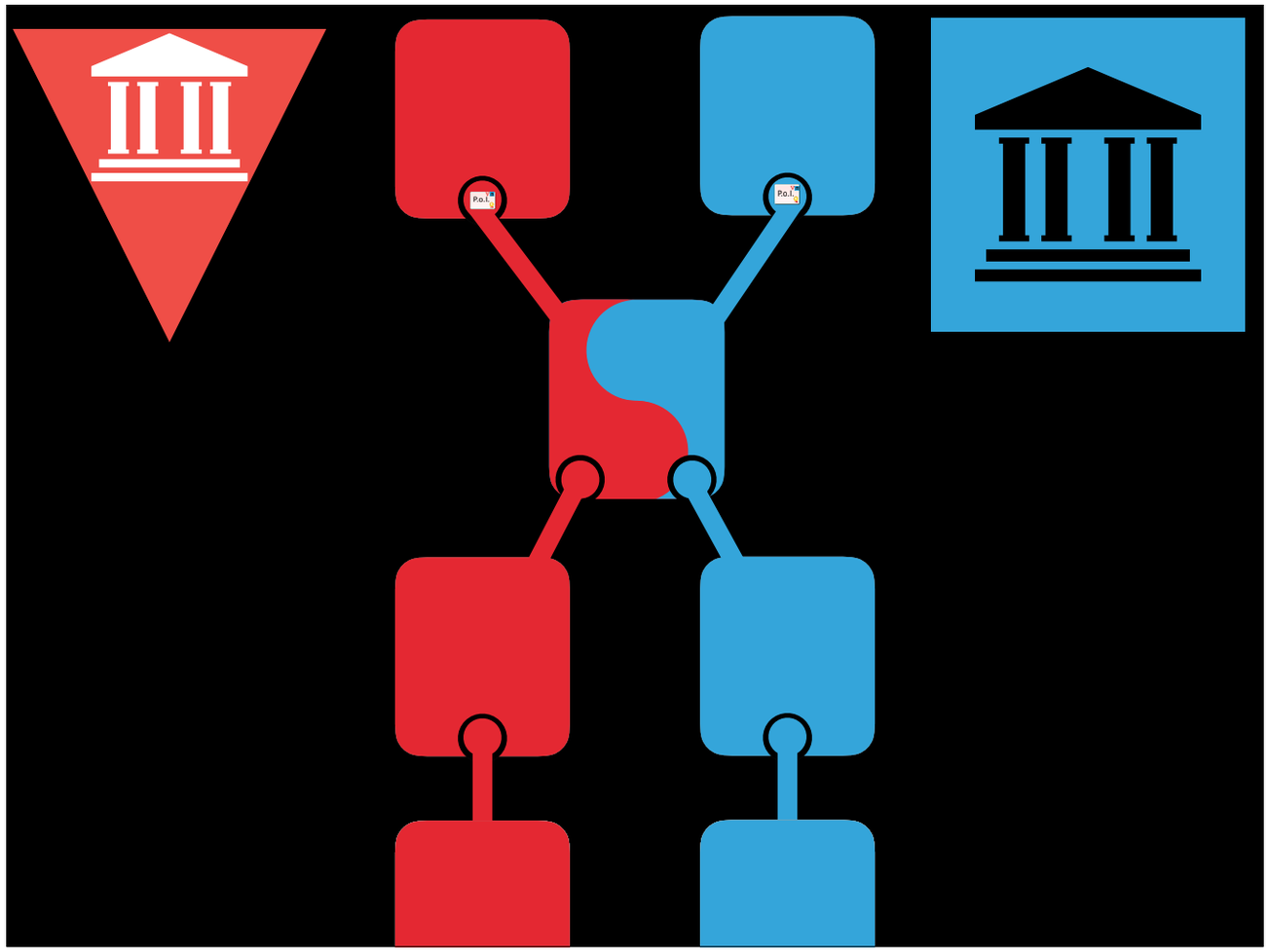
and then each bank’s Fern servers would approve it…



Once we have proofs of Integrity from each bank’s Fern servers, we combine them into one proof, with joined integrity…

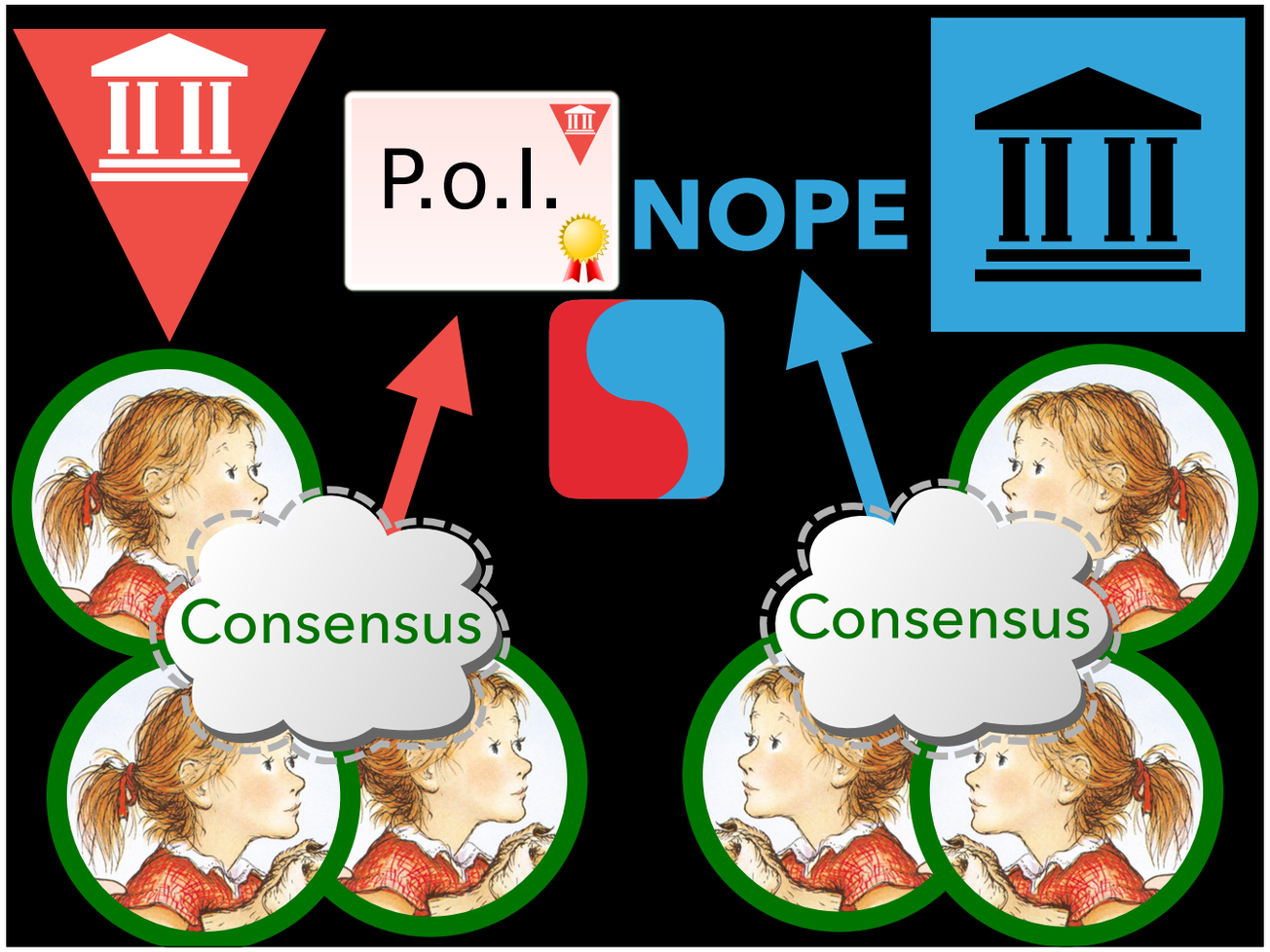


and then that becomes the proof which everyone should see whenever they reference this block, showing that it’s in both chains

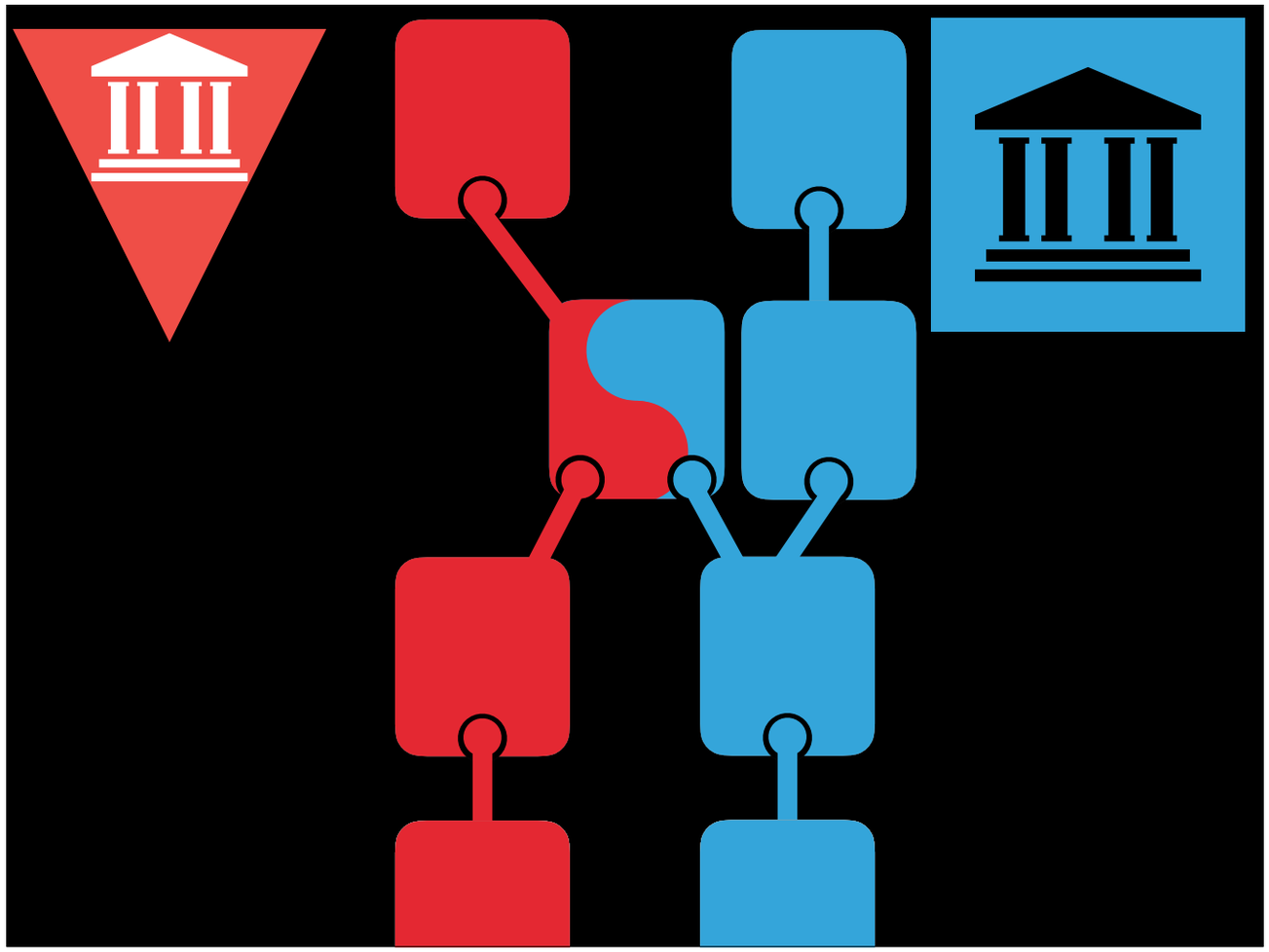


So in the continuing chains at both red and blue bank, this proof will be in the next blocks.

But there’s a problem with this naive plan…



Suppose that Red’s Fern servers agree to put this block in Red’s chain, but Blue’s Fern servers decide not to put this block in Blue’s chain.

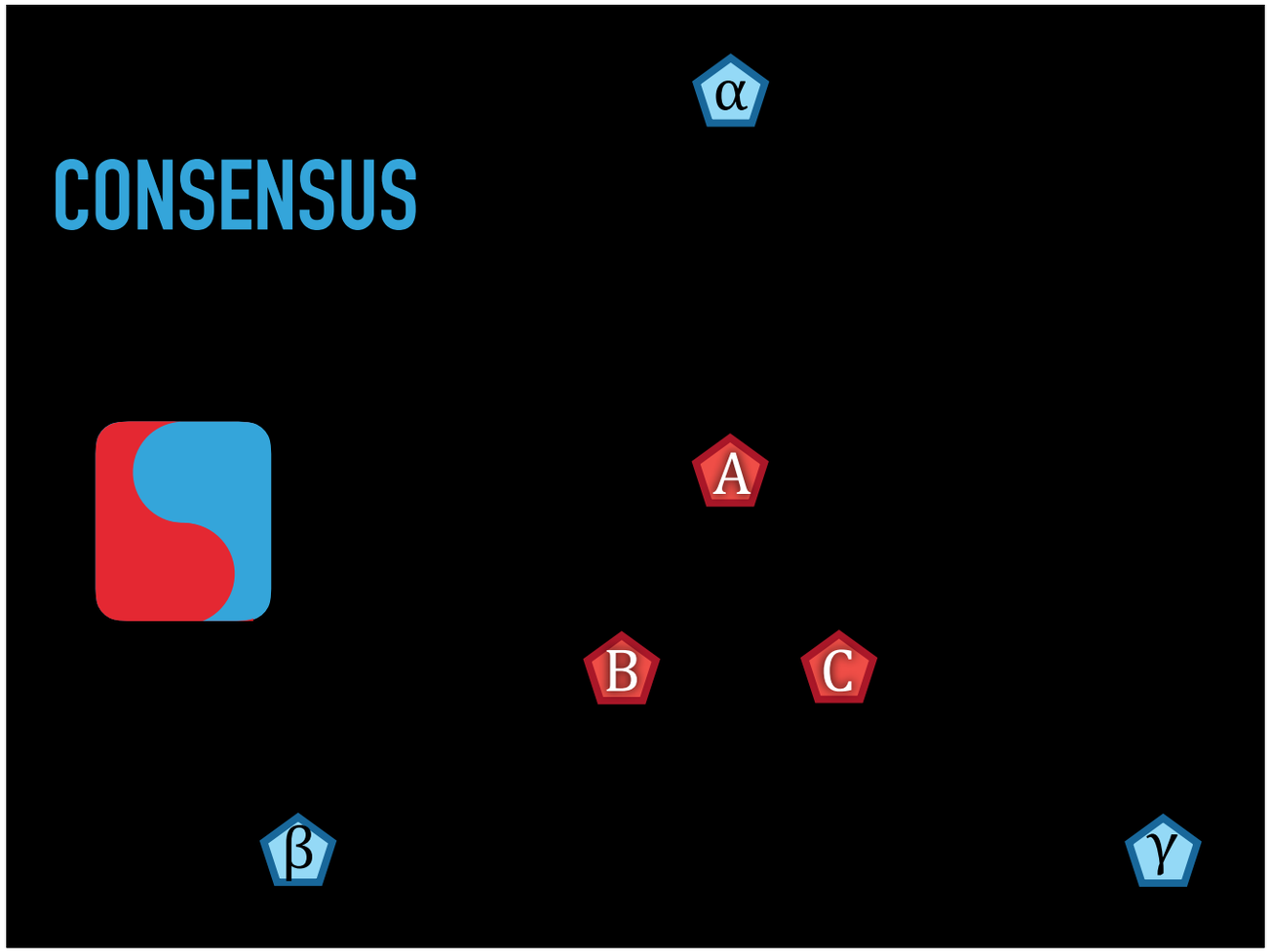


They might opt for something else instead, and so red's chain will include this block, but Blue's chain won't.

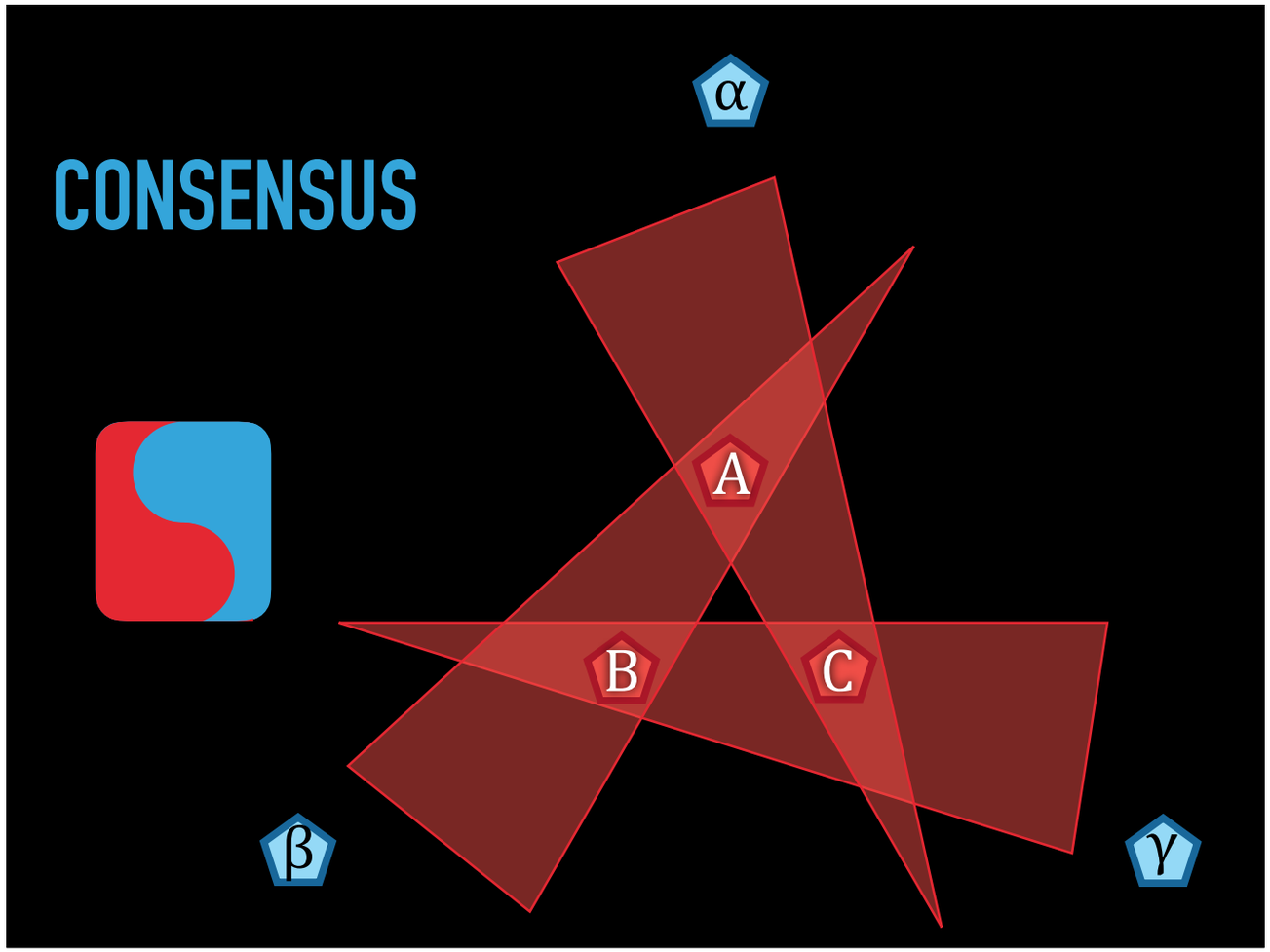
Anyone reading blue's state in the future would not read this block on their way down.

So Red and Blue need to figure out some kind of consensus they should demand of their conjoined block,
to prevent this kind of thing from happening.

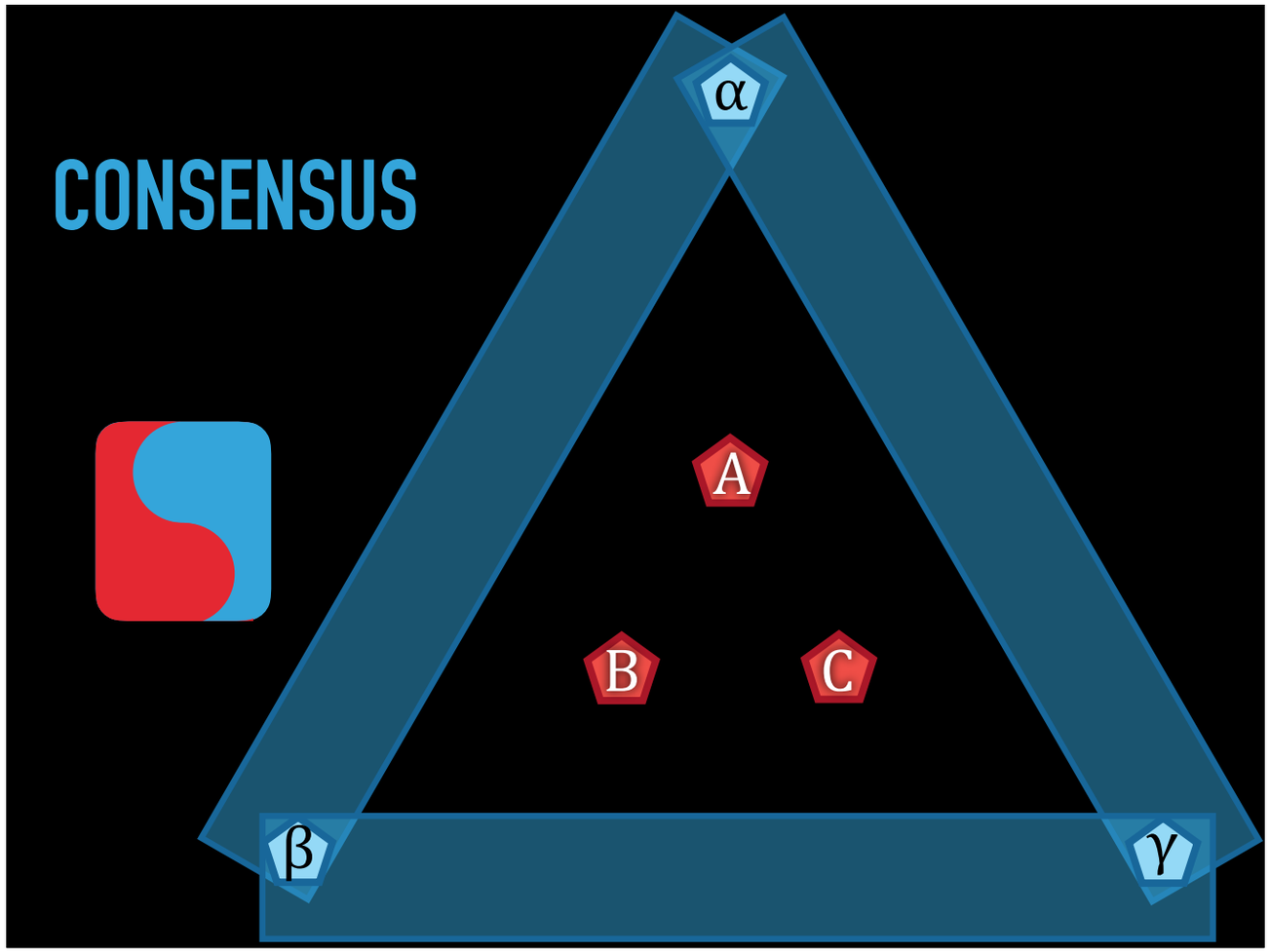
That brings me to something else which Charlotte enables: we're going to want more nuanced models of consensus, and algorithms to accomplish them for our Fern servers.



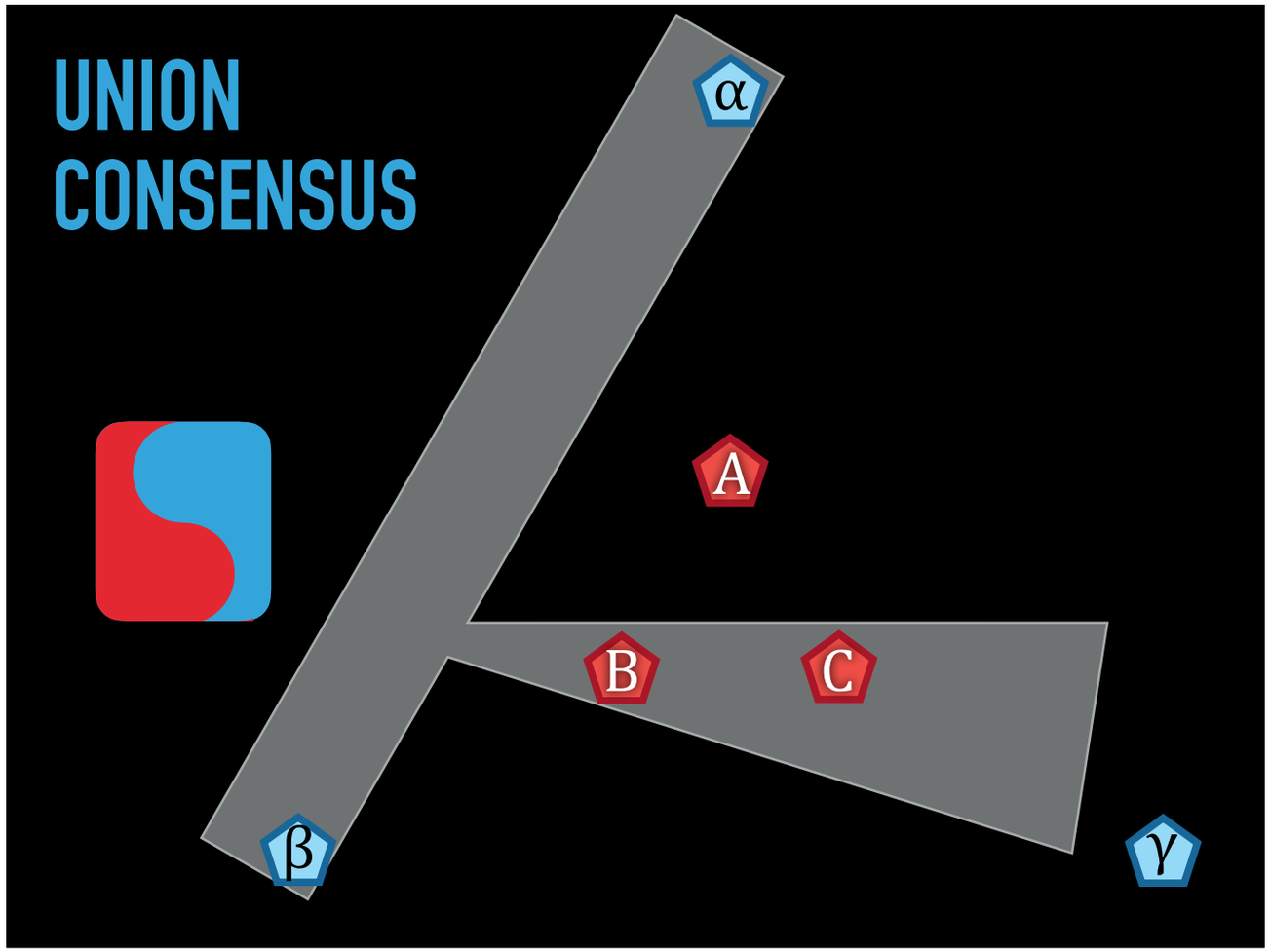
So, these are Red and Blue’s server groups,



and if you'll recall, Red's quorums look like this, which is to say, you need agreement from all of the servers in two of Red's server groups to commit a block.



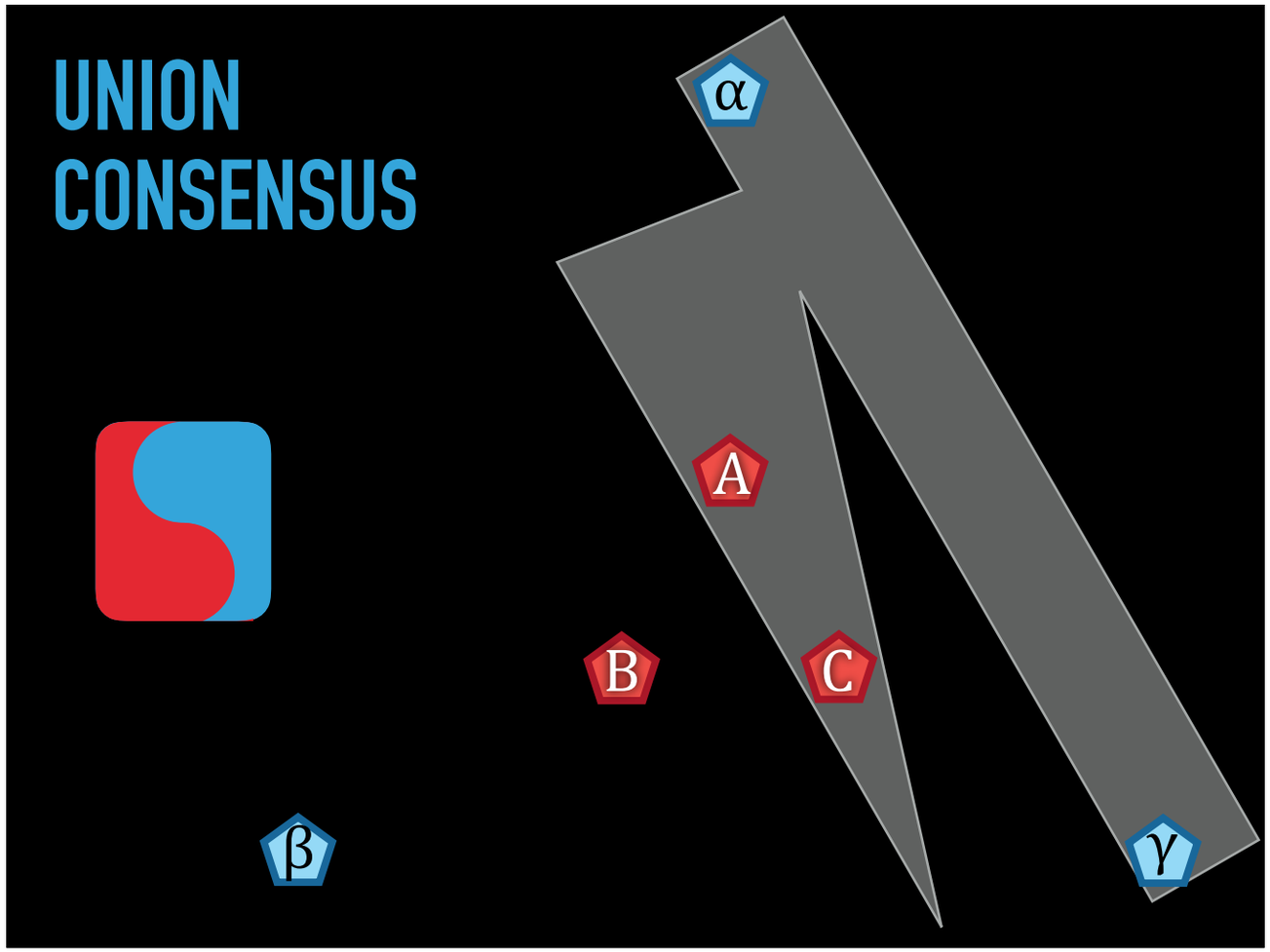
These are the quorums for blue.
you need agreement from all of the servers in two of Blue’s server groups to commit a block.



So one possible solution is a kind of “union consensus”: you need everything you’d need for Red, and everything you’d need for Blue.

That is to say, our quorums include all the servers from two Red groups, and all the servers from two blue groups.

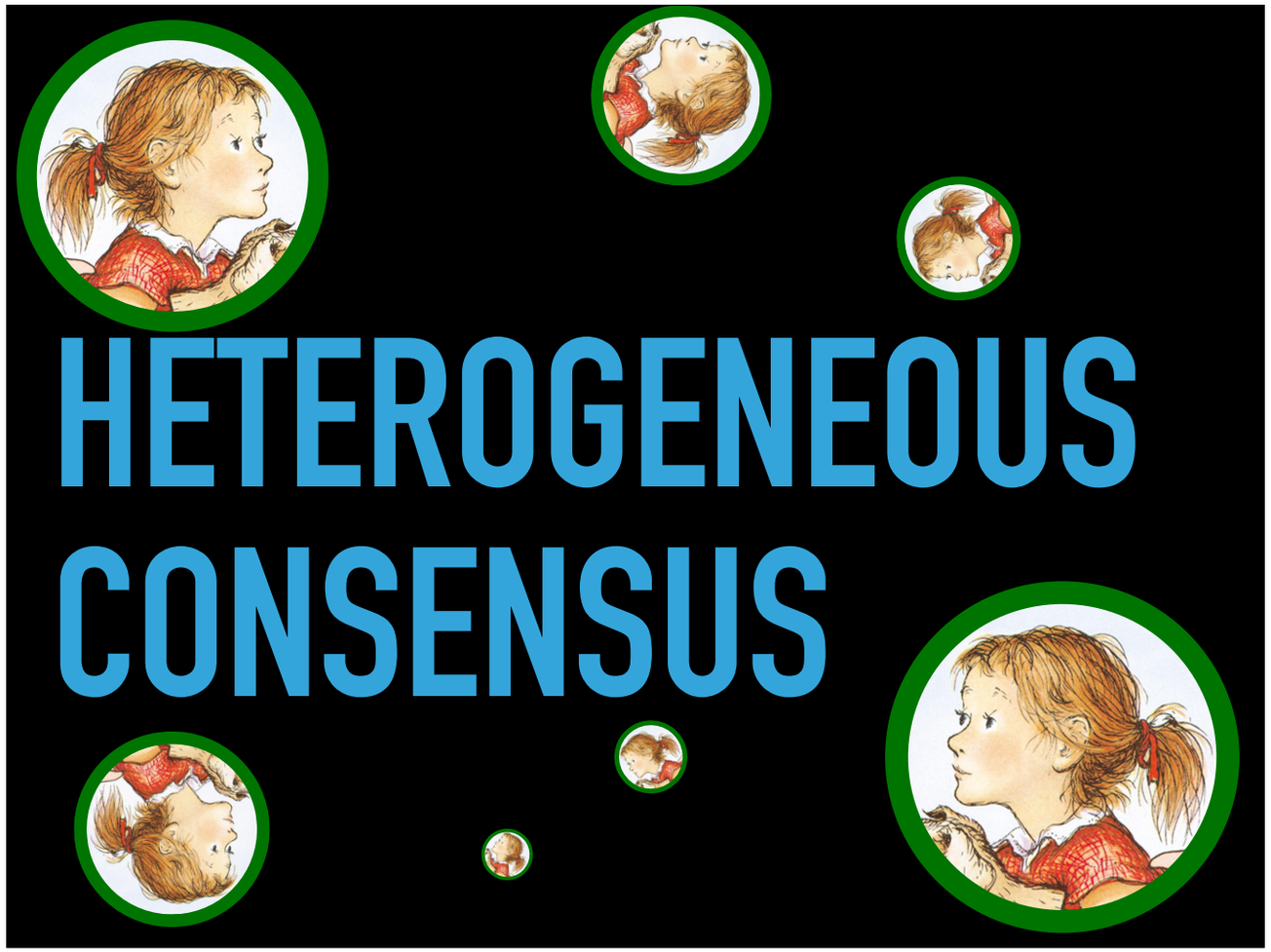
So this is a quorum…



and this is a quorum, and so on.

There are 9 of them, one for each pair of red and blue quorums.

Now all of this fits within the general Charlotte framework, but Charlotte says nothing about how all these servers actually perform consensus, it’s just a framework for what to do once you’ve got some attestation of consensus.



And this is where an earlier project of ours fits into the Charlotte paradigm:

heterogeneous consensus, which is a more general formalization of asynchronous consensus, complete with a protocol for achieving it, and a proof of concept implementation we’re using in our proof of concept Fern servers.

2 BANKS EXAMPLE

MORE COMPLICATED CONSENSUS

- Neither bank permits the other to block transactions
- Agree when possible
- Bank's assumptions about its own servers guarantee non-equivocation

By way of motivation, suppose Red and Blue banks are somewhat more paranoid.

Suppose they want to agree on blocks together, but neither trusts the other's servers not to D.o.S. the system.

Of course, they still want to agree when possible, and perhaps more importantly, each bank should never ever be able to equivocate:
as long as a bank's assumptions about its own servers are met, it should never commit to conflicting values.

Now, it turns out that if you think about it for a while, this means there have to be cases where bankers at Red bank think something different than bankers at Blue bank.
For instance, if all of Blue bank's servers are unresponsive, it turns out Blue bankers can't decide anything, but Red bankers can.

HETEROGENEOUS CONSENSUS

- Neither bank permits the other to block transactions
- Agree when possible
- Bank's assumptions about its own servers guarantee non-equivocation

This is where heterogeneous consensus comes in: we want to characterize exactly what's possible, and when.

But let me explain what I mean by heterogeneous.

3 KINDS OF HETEROGENEITY

There are 3 kinds of heterogeneity that are relevant here.

HETEROGENEOUS CONSENSUS

3 KINDS OF HETEROGENEITY

- Heterogeneous Failures
  - "mixed failure model"

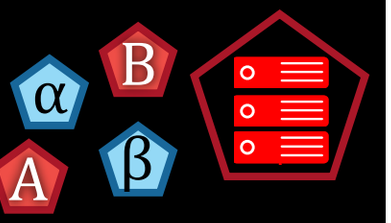
Firstly, there are heterogeneous failures:
not all failures are the same,
there's a difference between crash and byzantine.

In fact, we've been making mixed failure model assumptions all along: no entirely byzantine server group, no crashes in more than one group
see, we're a little bit heterogeneous already!

HETEROGENEOUS CONSENSUS

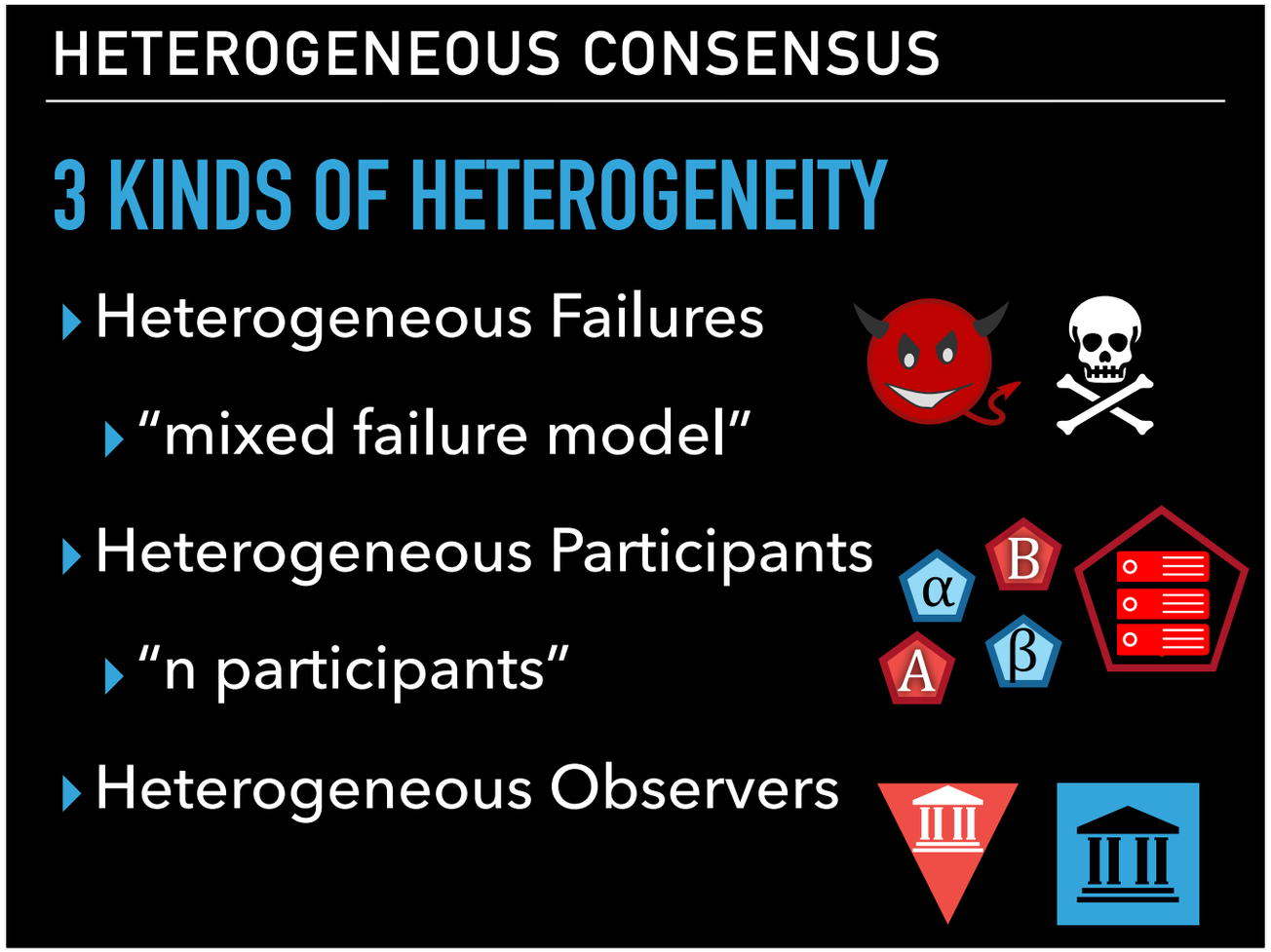
3 KINDS OF HETEROGENEITY

- Heterogeneous Failures
  - "mixed failure model"
- Heterogeneous Participants
  - "n participants"



Secondly, not all participants are created equal.
In a lot of traditional systems, the only relevant number is the number of participants.
But we've got richer assumptions, about groups of servers and who owns them.

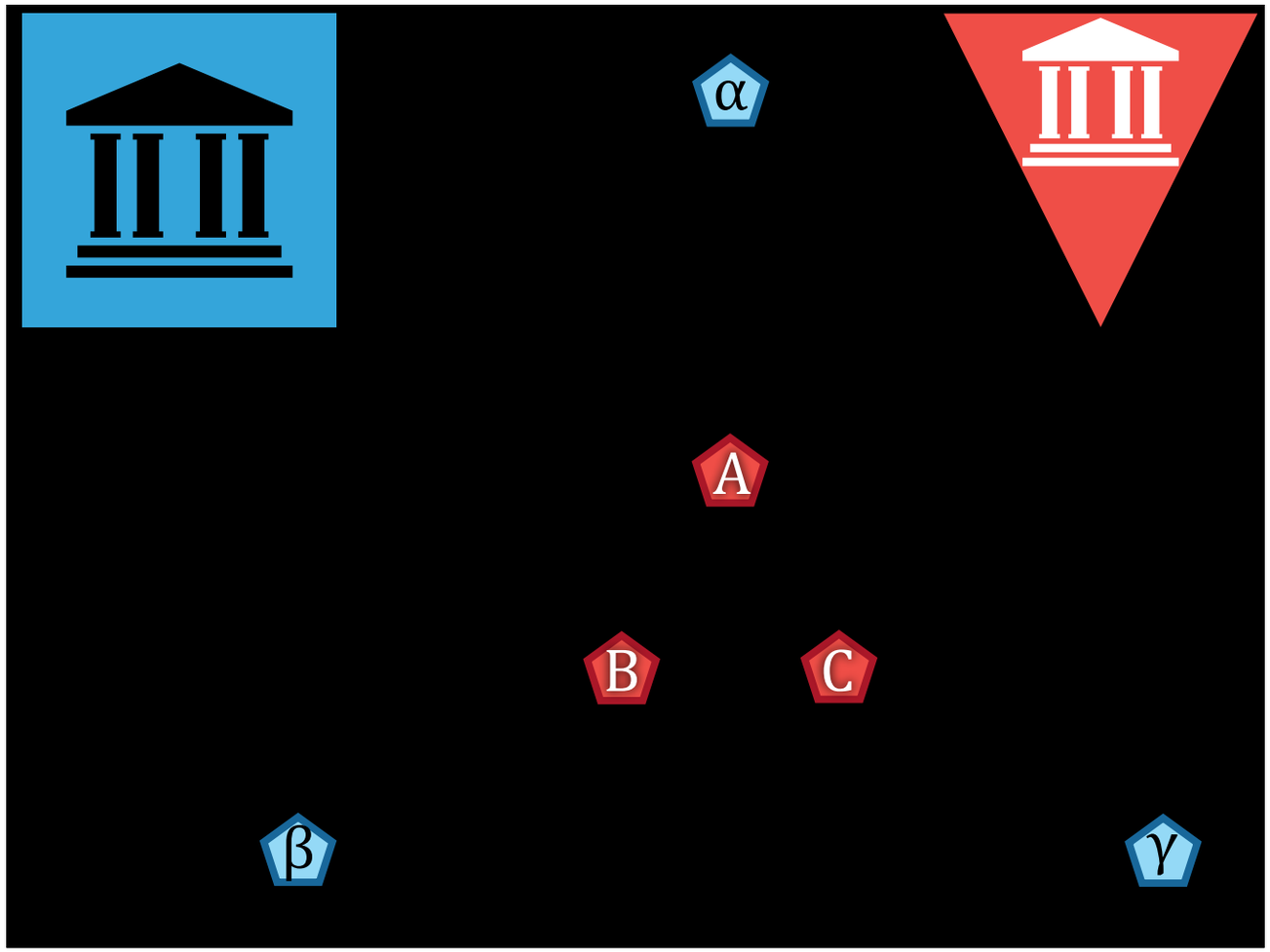
That's another kind of heterogeneity.



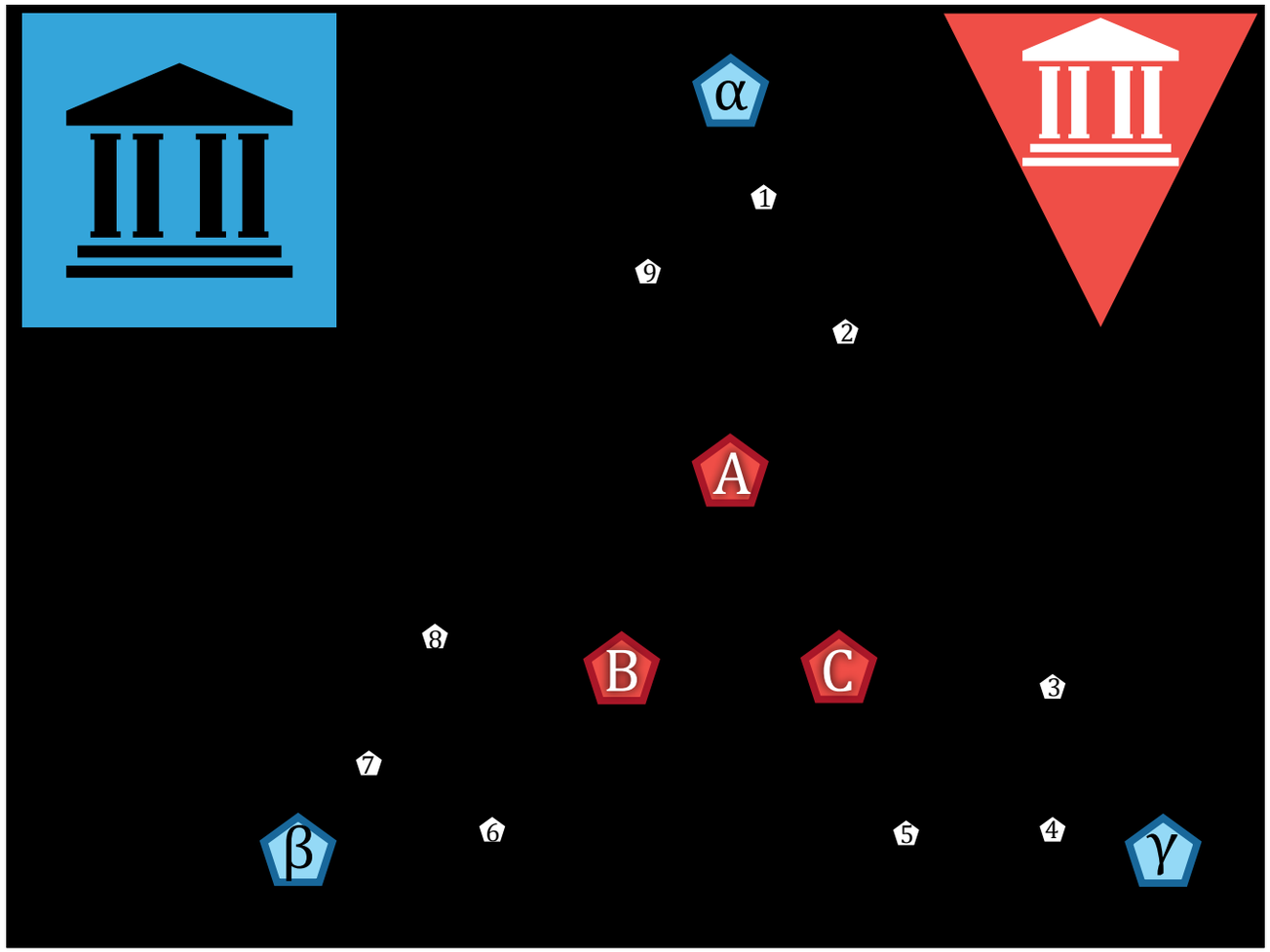
And finally, not all observers are created equal.

Observers are an often-overlooked part of distributed systems,
but ultimately, someone is learning the values we're consenting on: someone looks at the output from all the participants and "believes" or doesn't believe that a block belongs on a chain.

In this case, they're the bankers!
The bankers at Blue bank believe one thing, and it's not necessarily the same as the bankers at red bank believe.



Now, it turns out that what red and blue are asking is impossible with only these servers.
Basically, neither trusts the other not to lie to them, and each could decide one thing on its own while telling the other it did the opposite.

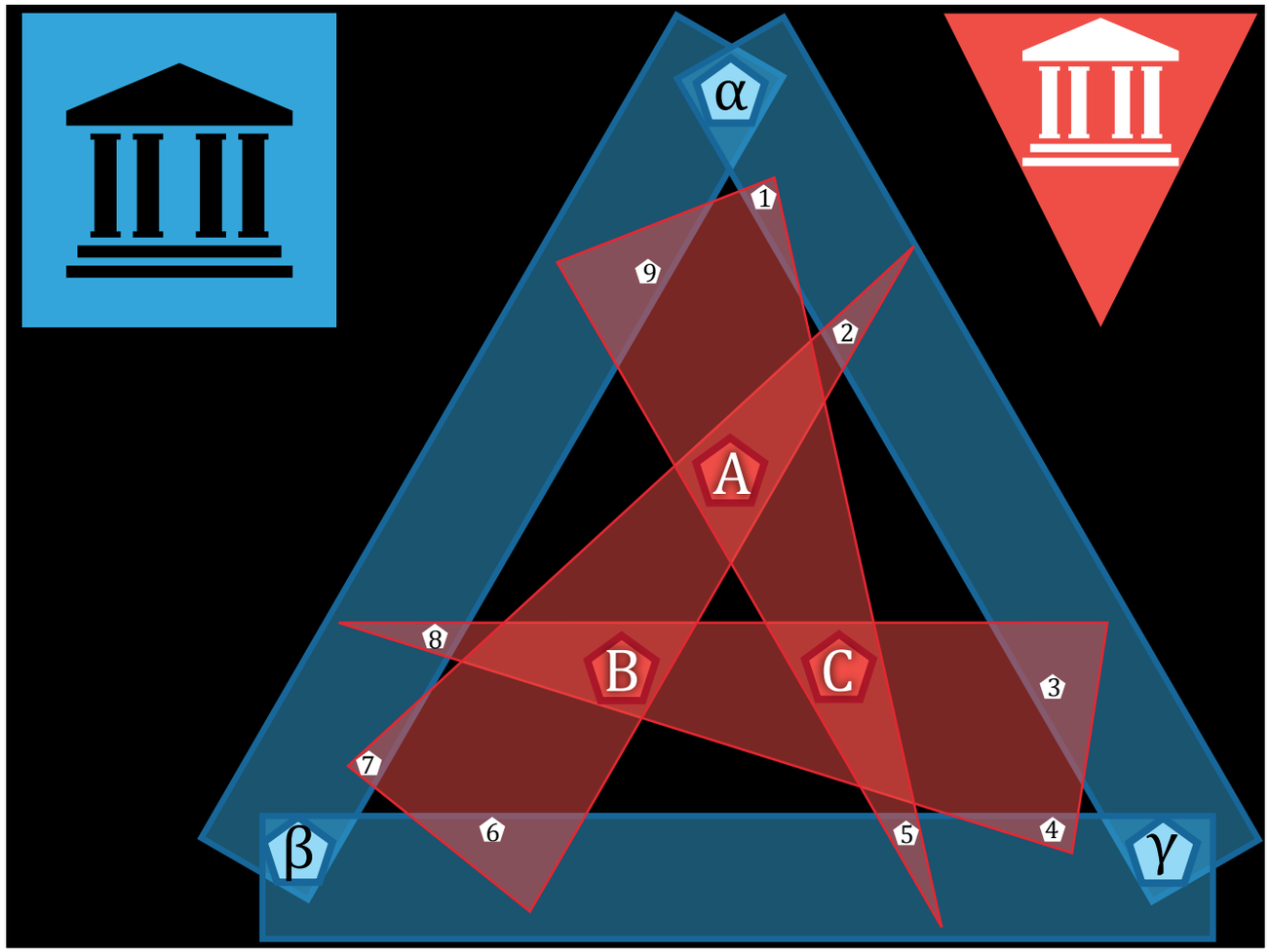


So we have to sprinkle in some third parties.

Suppose these, too are groups of servers.

Suppose we trust these a little less.
Each bank still demands that it can’t equivocate internally unless its wrong about its own servers’ failure assumptions,
so the worst that a third party group can do is make the two banks disagree, if that group is entirely byzantine.

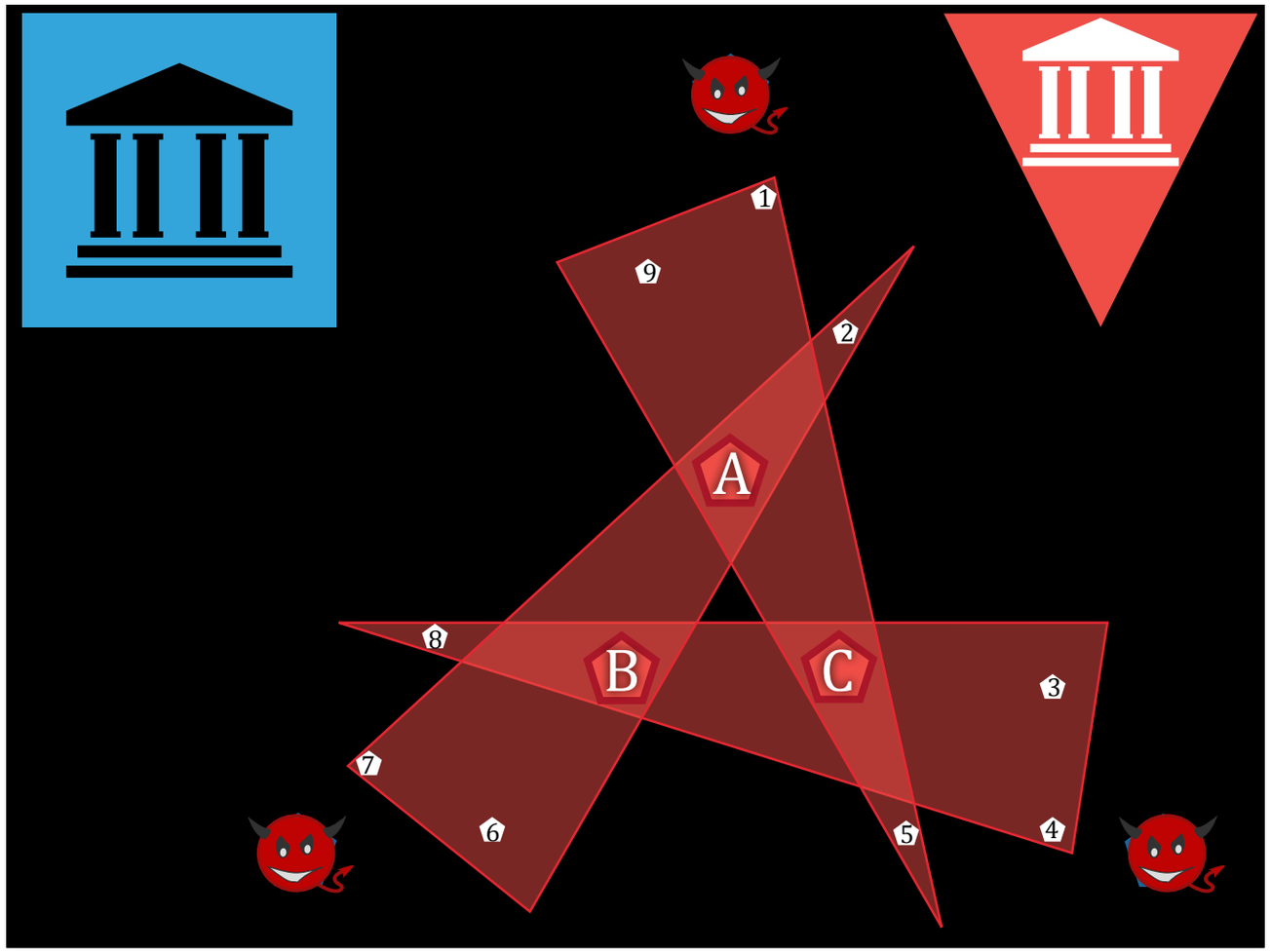
It actually takes 3 third party groups crashing to D.o.S. a bank.



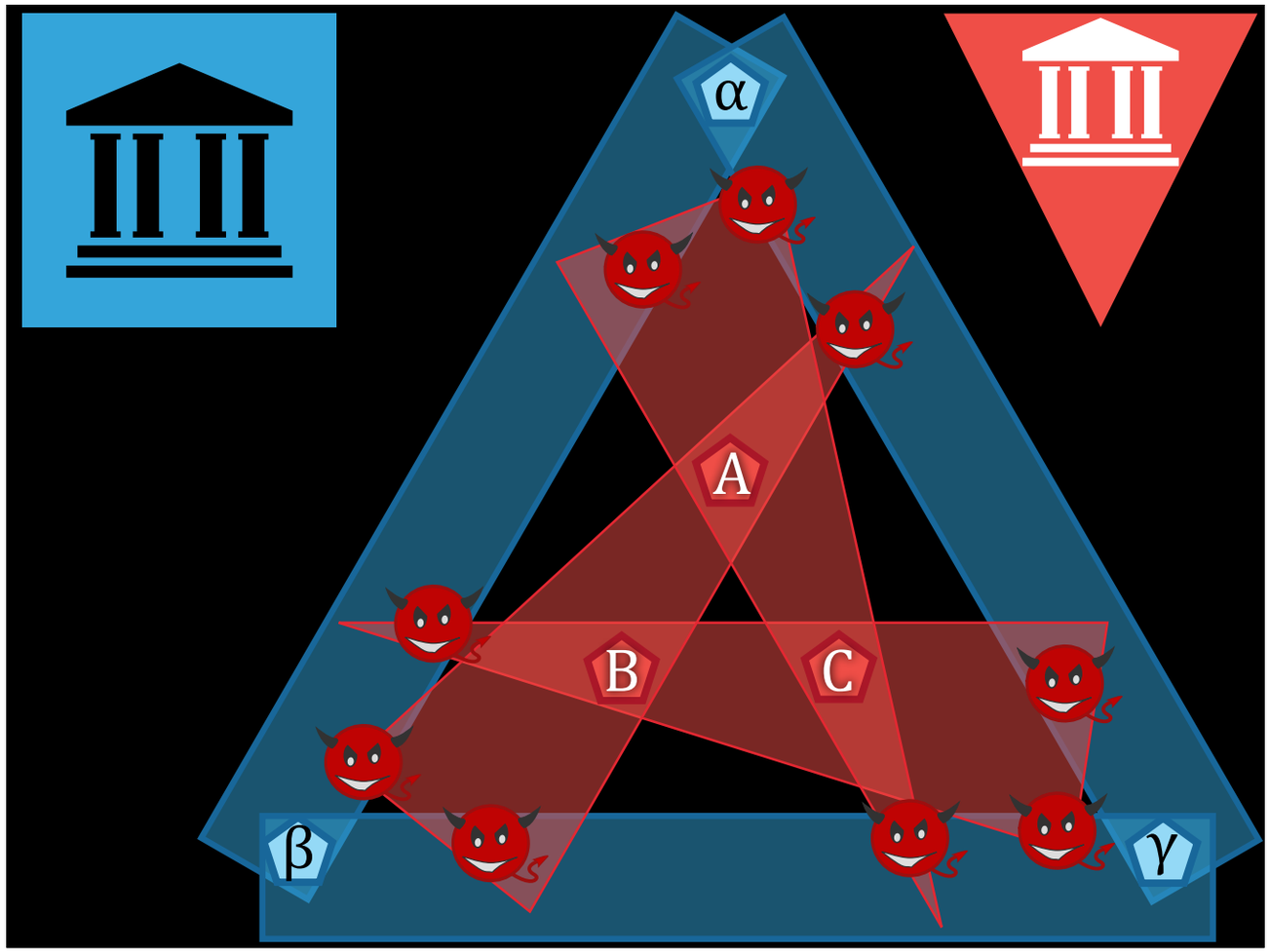
So here are the bank’s quorums.

Any red quorum is sufficient for red to decide

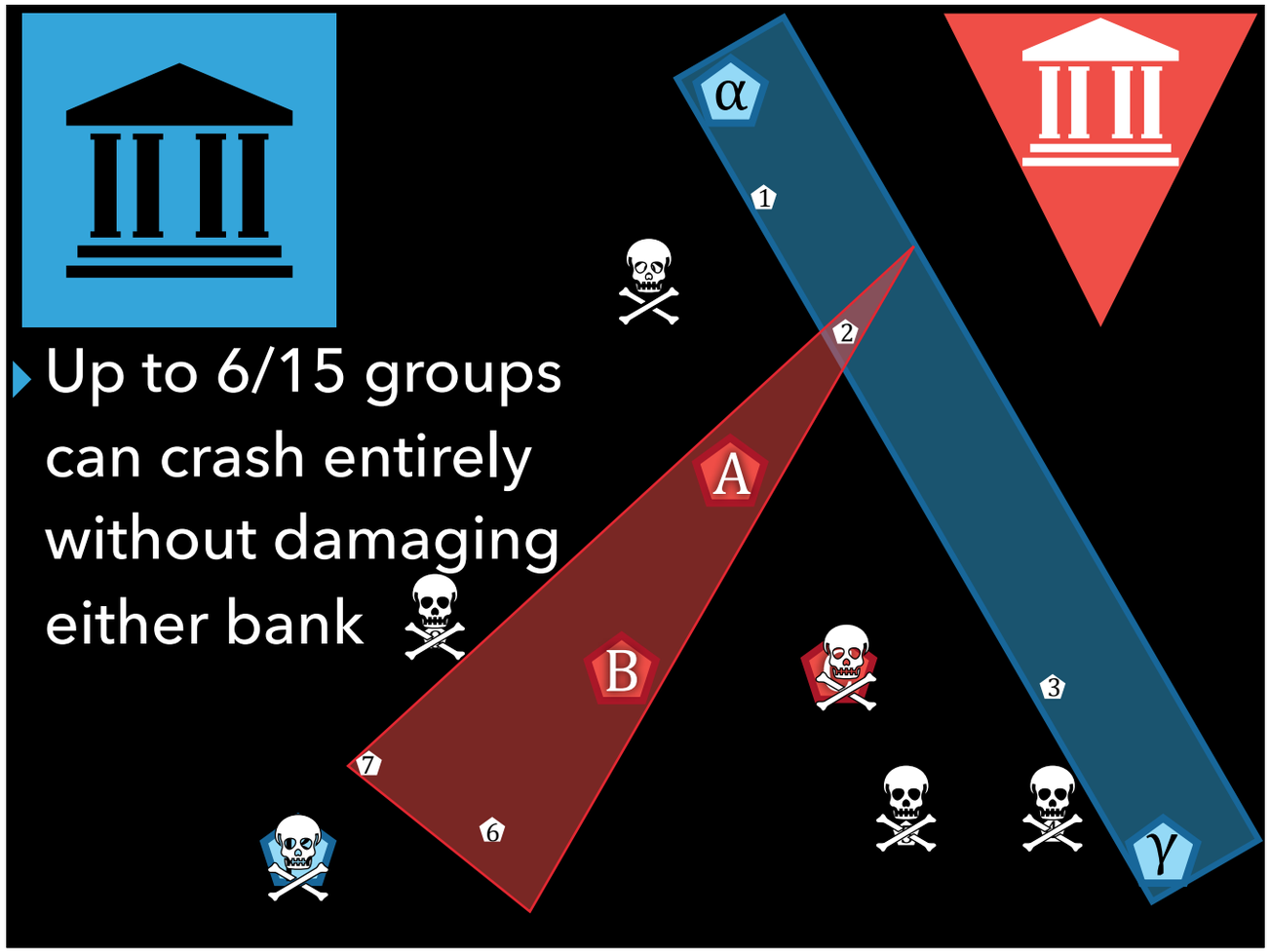
and any blue quorum is sufficient for blue to decide



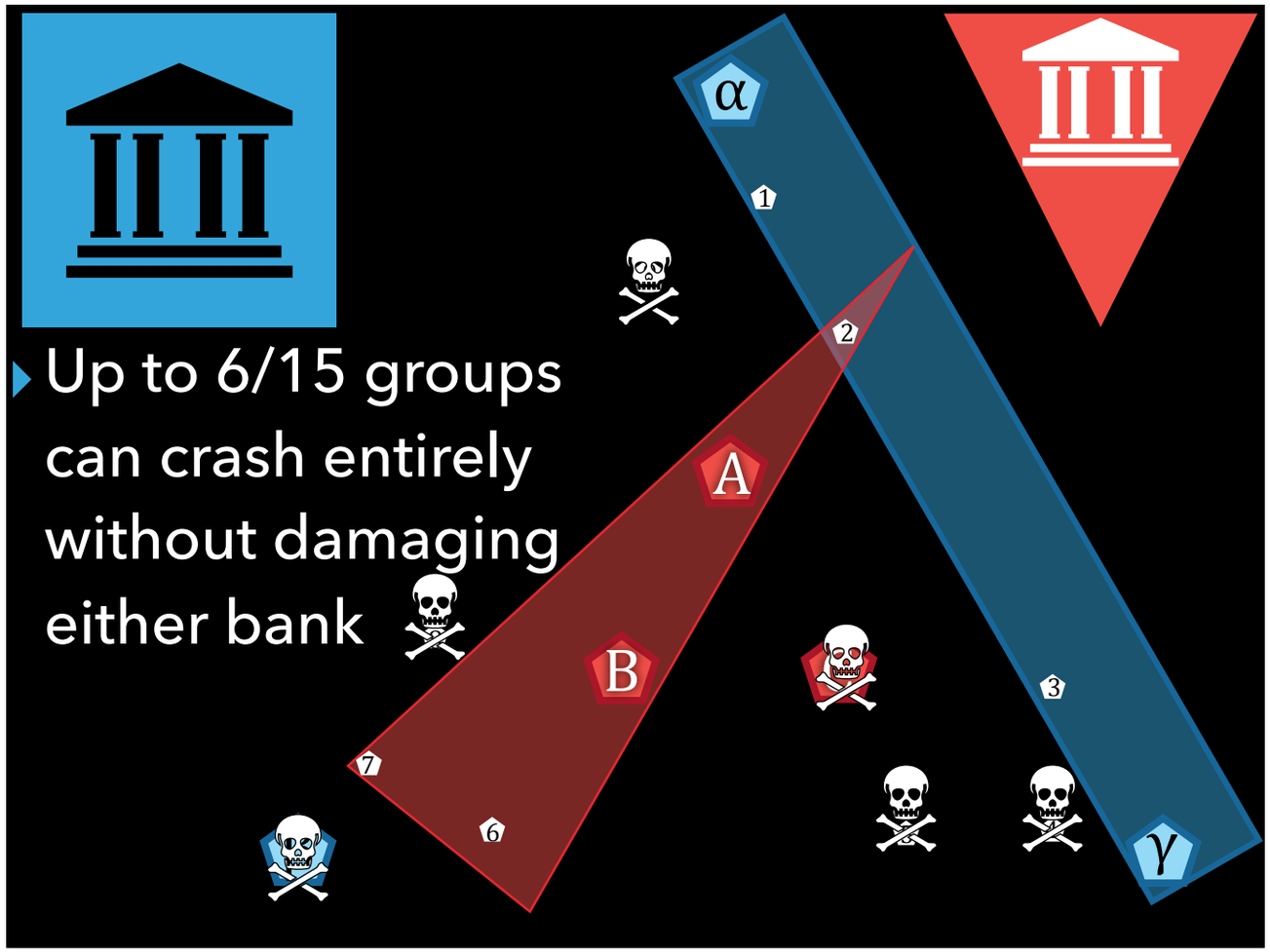
Even if one bank is entirely byzantine, it doesn’t affect the other bank at all.



Of course, if the third parties are byzantine, there’s no guaranteeing the banks agree, but internally, the banks won’t equivocate

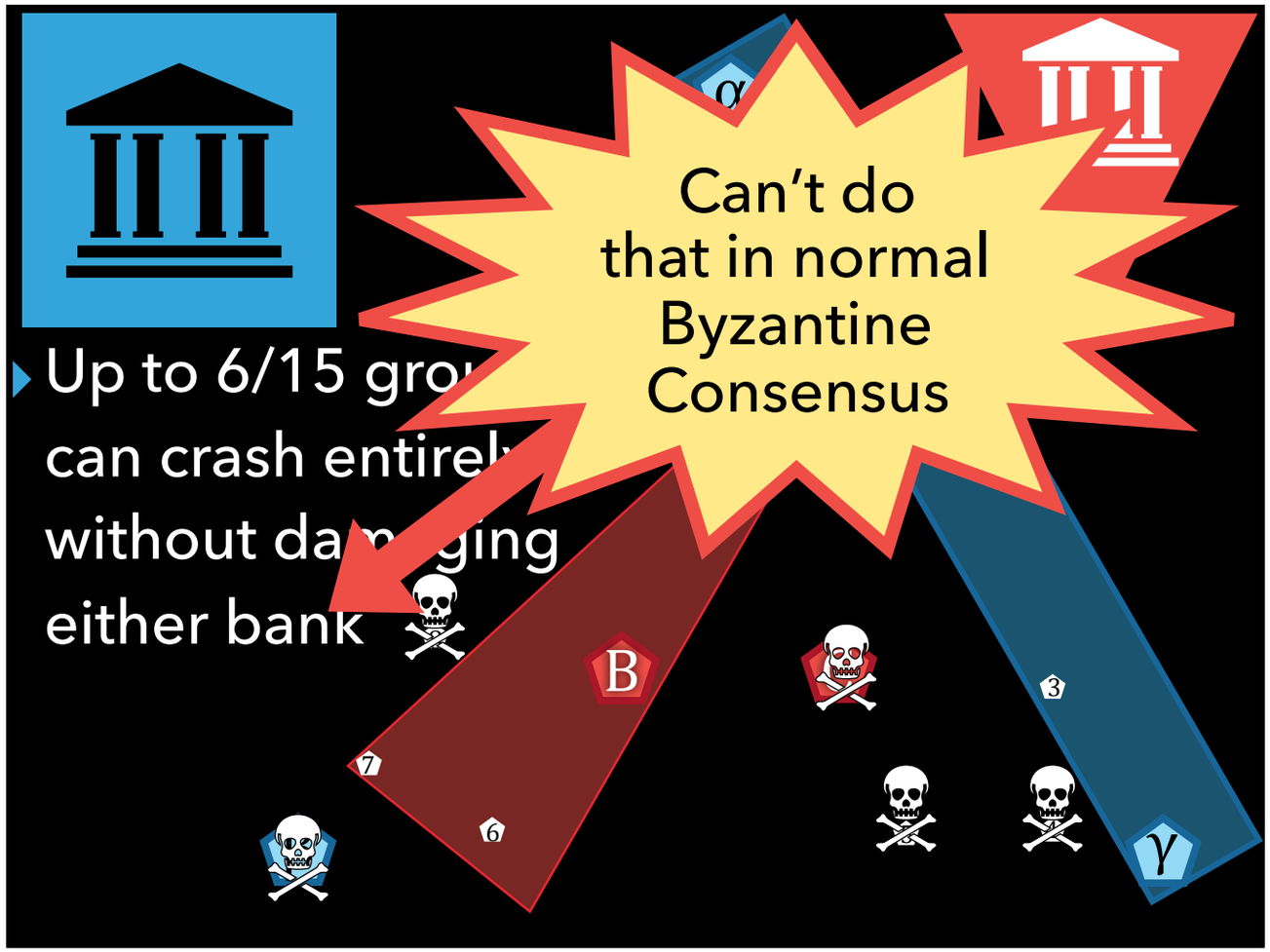


and we can actually crash up to 6 of the 15 server groups without killing liveness for either bank.



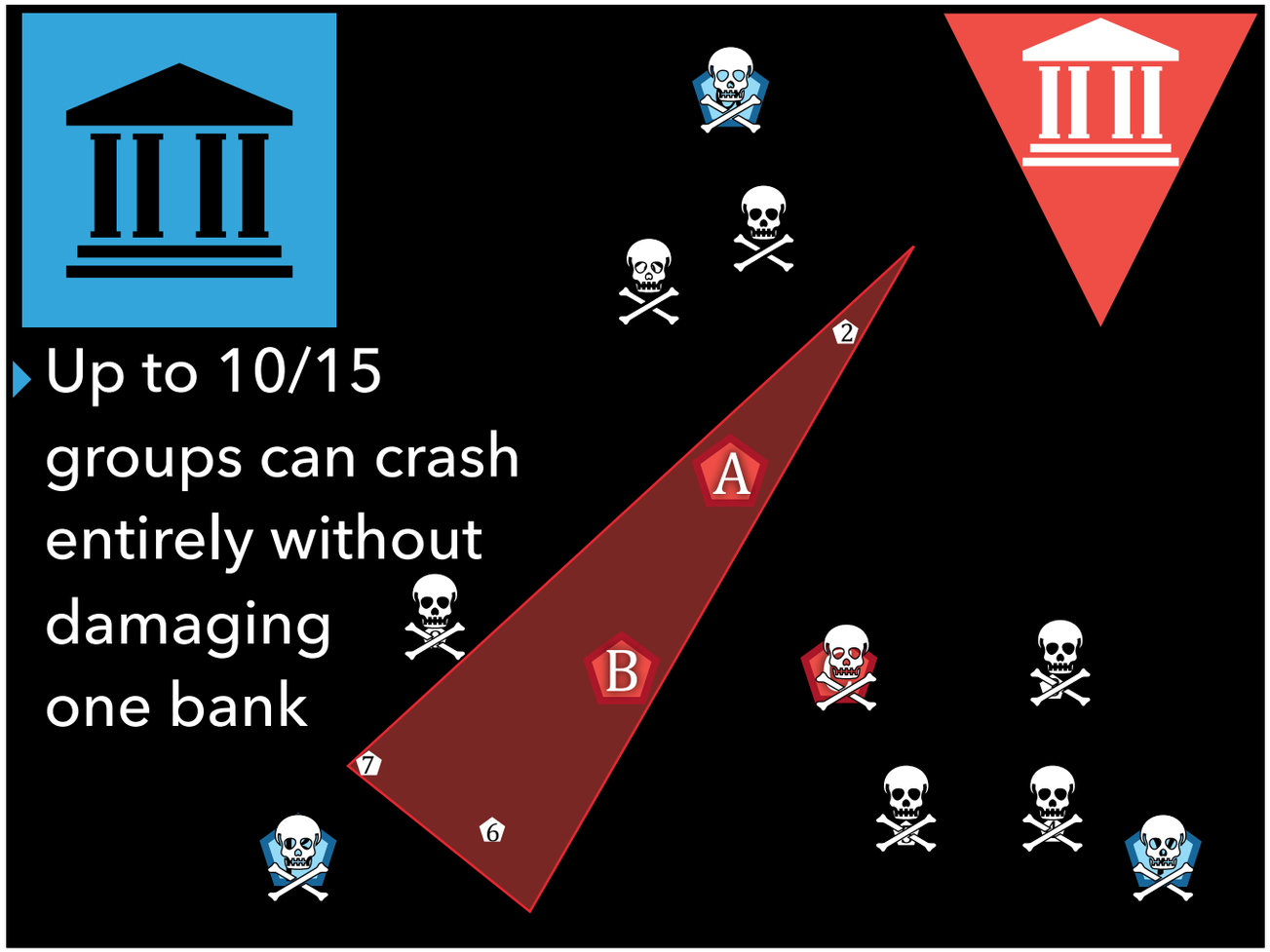
Which isn’t even possible in traditional byzantine consensus.
It’s more than 1/3 failures.

Our more nuanced assumptions give us more failure tolerance.

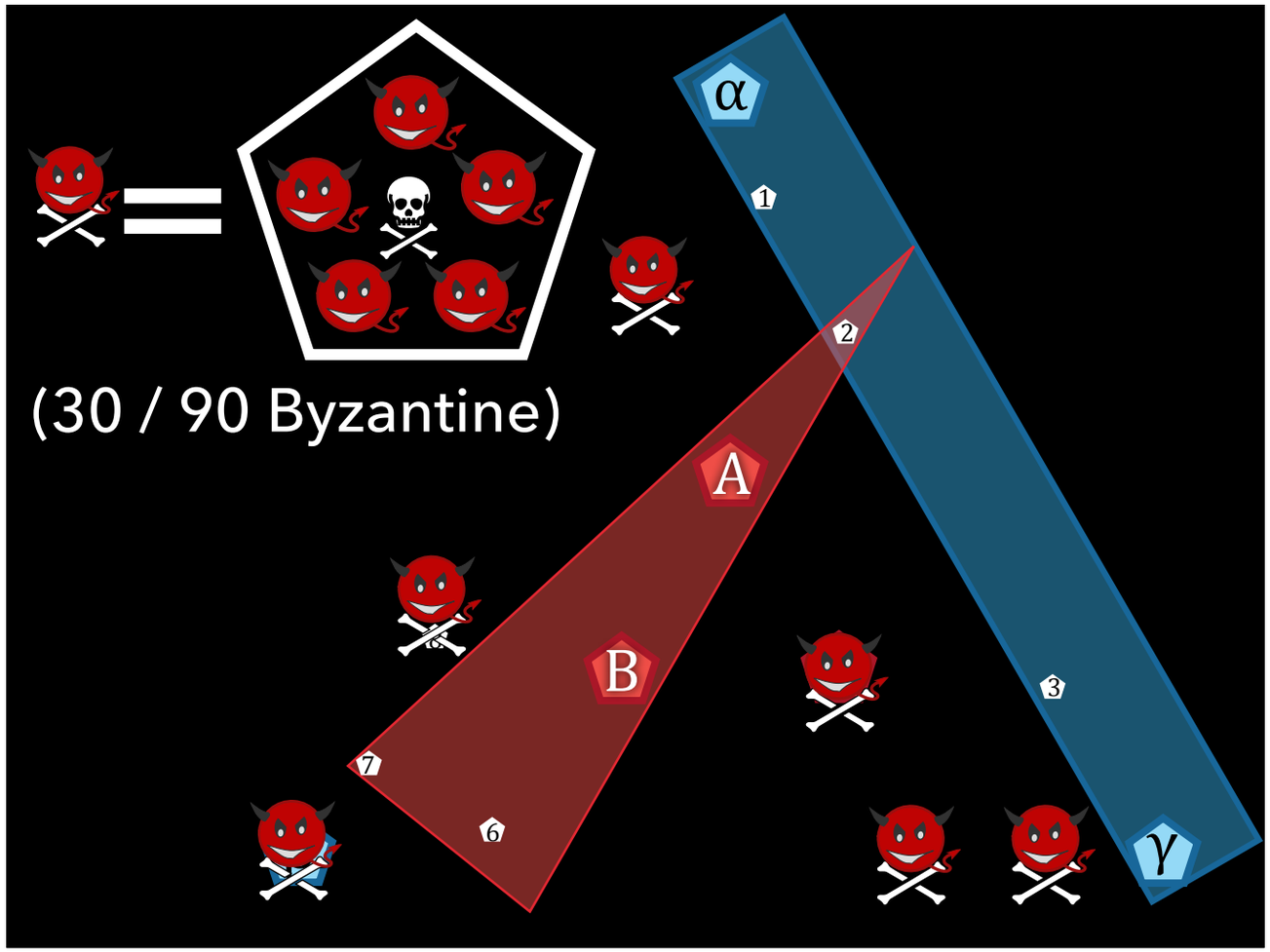


Which isn’t even possible in traditional byzantine consensus.
It’s more than 1/3 failures.

Our more nuanced assumptions give us more failure tolerance.

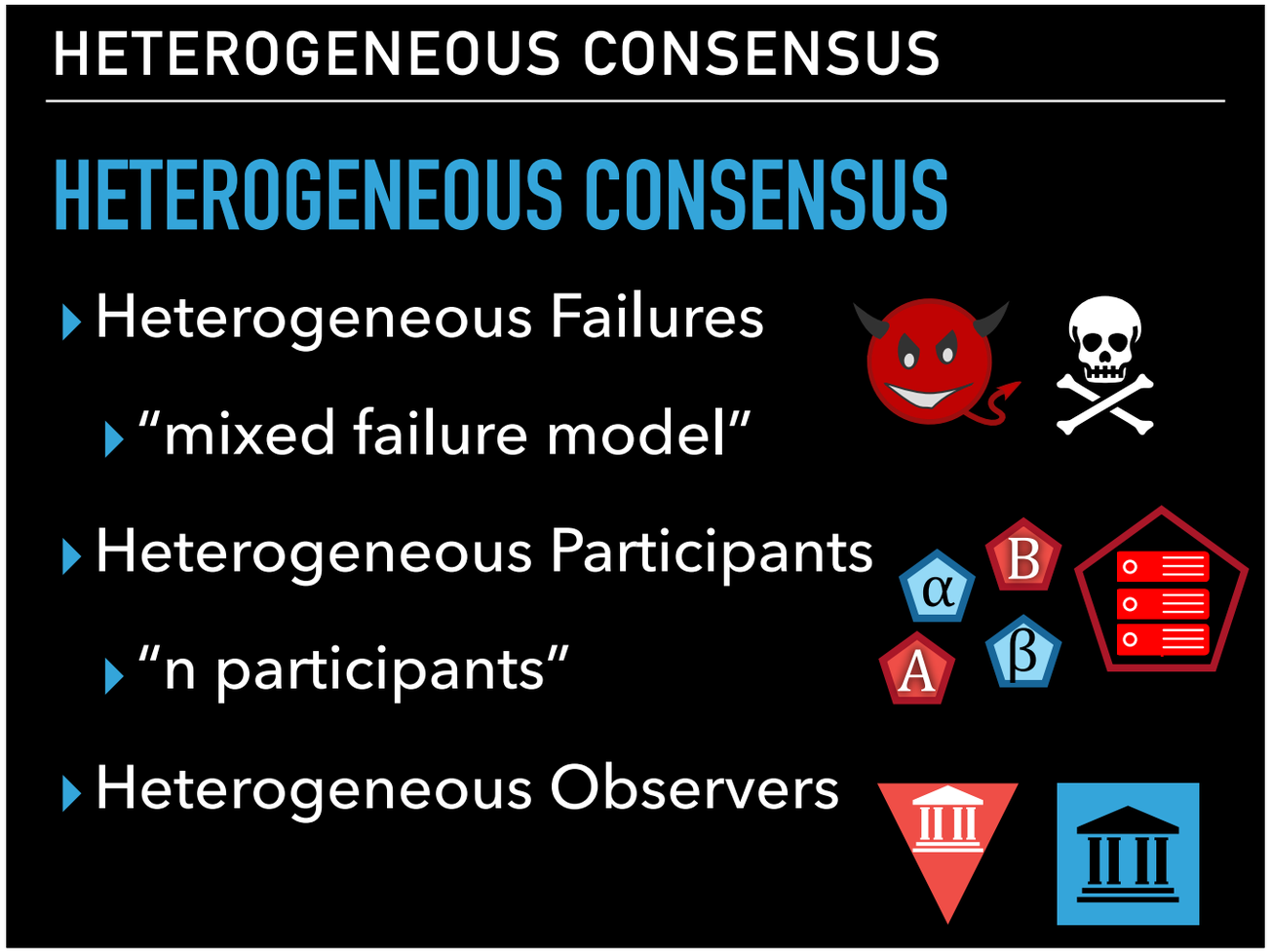


Heck, we can crash 10 groups while keeping one bank totally online.



There’s an extreme case I find amusing
Suppose each server group has 6 servers, and
for these ones, there’s one crashed and 5 byzantine in each.

Now there are 30 byzantine failures without hurting either bank
That’s a full 1/3 byzantine failure, which isn’t survivable in any traditional (asynchronous) technique.

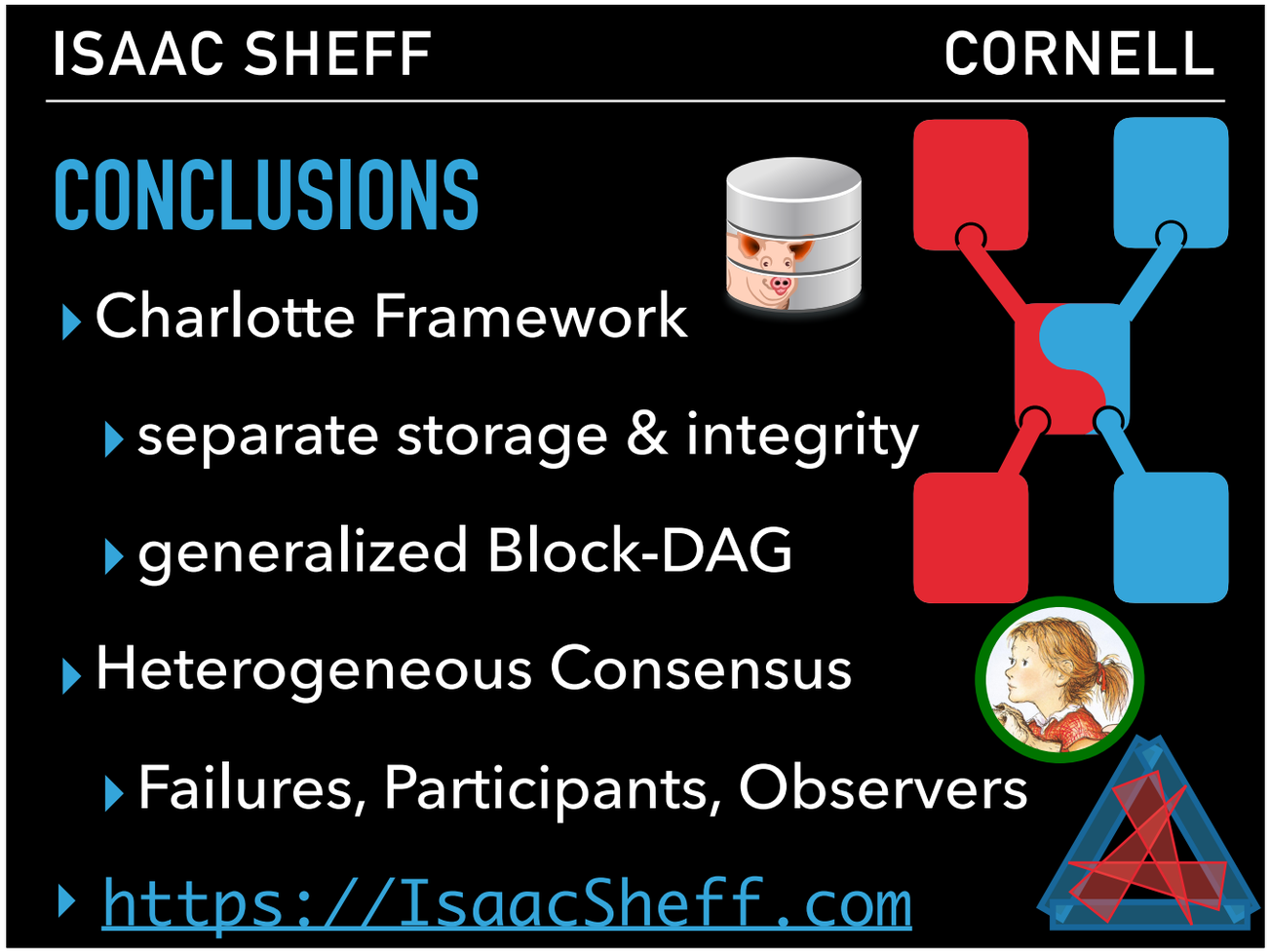


Anyway, that's just an example of what heterogeneous consensus can do.

we've also formulated conditions for when consensus is actually possible,
because, for instance, it's not possible if you want to agree when no participants are live.

We've got a working formalization, paxos-based algorithm and implementation, where

Not all failures are created equal
Not all participants are created equal, and
Not all observers are created equal



In any case, those are our projects.

I'd love to hear feedback, especially on the charlotte project, where again
our key idea is to separate storage and integrity, to
allow for generalized data structures within the distributed block-DAG,

and I got to touch on Heterogeneous consensus as well,
which is heterogeneous in failures, participants, and observers.

You can find these slides, and some possibly helpful links, on my website.
Thanks!