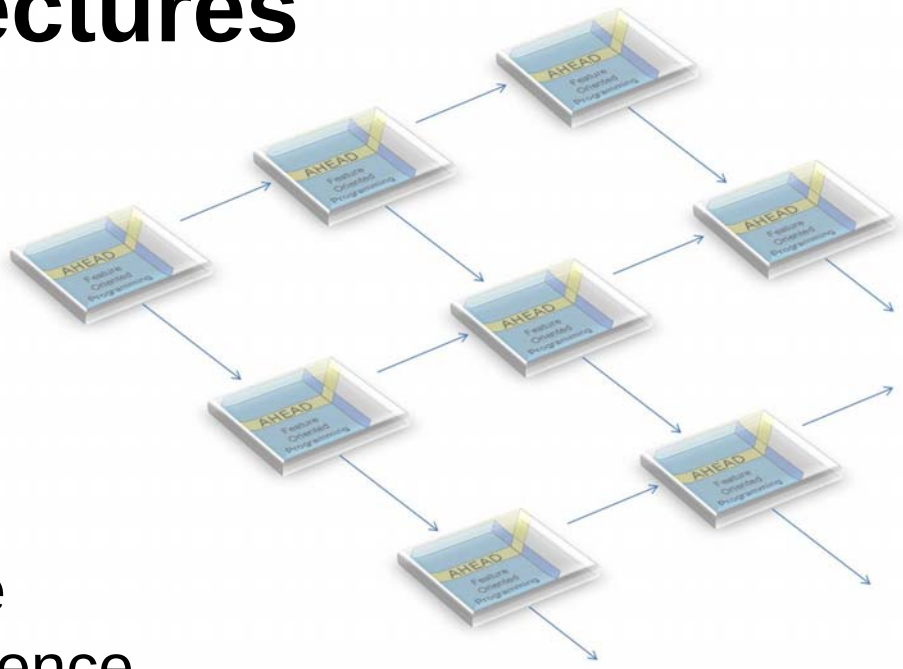
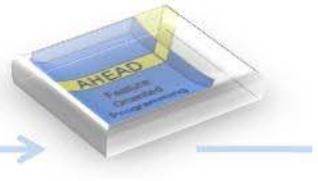
Refinement and Optimization of Streaming Architectures



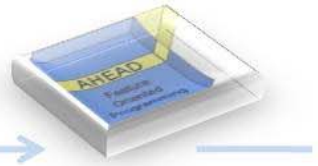
Don Batory and Taylor Riché
Department of Computer Science
University of Texas at Austin

Introduction

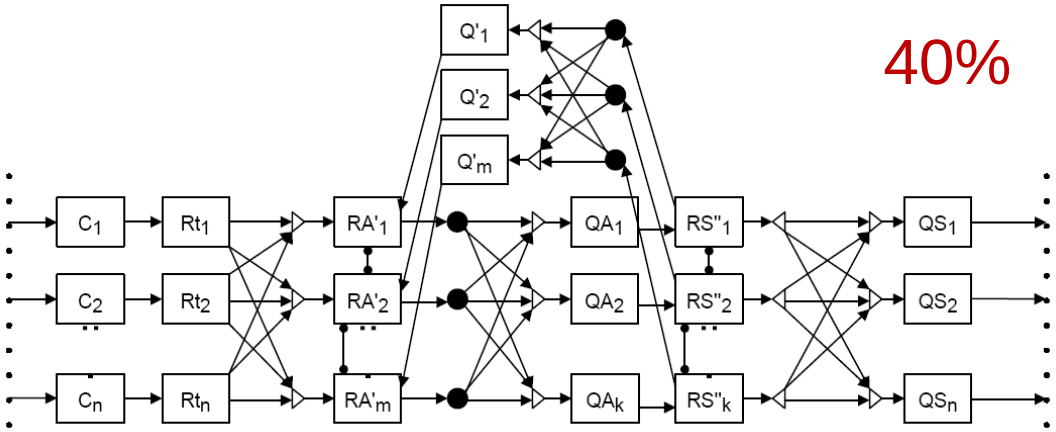


- **Model Driven Engineering (MDE)** is paradigm that will be increasing important to SW design & development
  - focus on problem essentials, not accidental complexity
  - frees us from low-level programming languages, platforms
  - way of the future…. with some interesting implications
- Today's presentation: a different view on MDE
  - not the way MDE is practiced today
  - show how *sequential architectures* (models) can be mapped to *parallelized or crash-fault-tolerant architectures* (models) in a way that can be explained, verified, built, and tested in a systematic manner

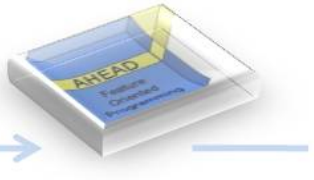
Streaming Architectures



- **Byzantine Fault Tolerant (BFT)** and **Recoverable Crash-Fault-Tolerant (RCFT) Servers**
  - reimplementing designs of domain-experts to automate their work
  - architectures of these servers are so complicated, we had to find a way to convince ourselves of the correctness of our designs
  - small handful of experts know how to build these servers
    - » about ½ seem to be at UTexas ☺
  - *need systematic way to help explain, verify, build, and test*

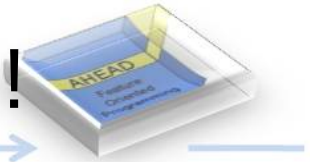


Stepwise Development



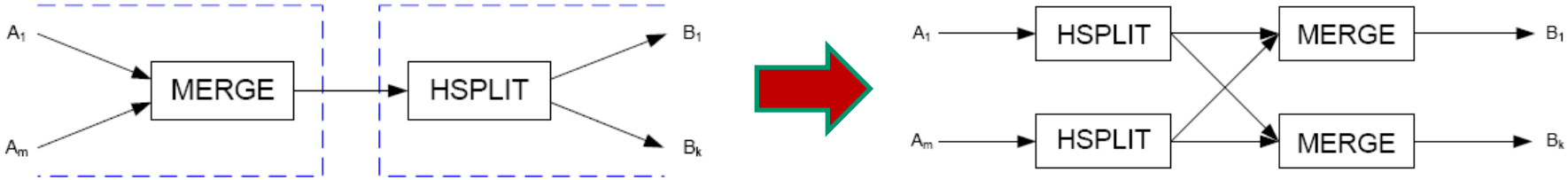
- Classical & fundamental way to control design complexity
  - our work builds on results of a long line of pioneers (Labview 80, Gorlick 92, Broy 92, Moriconi 94, Garlan 96, Rumpe 97, Kong 03, Clarke 06, …)
  - use a standard component-connector model of application
  - elaborate it by simple transformations
  - use standard notion of hierarchical refinements (transformations)

Hierarchical Refinement is Not Enough!

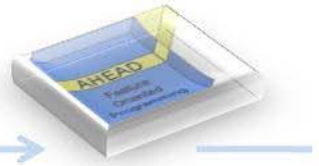


- *Controversial*: adding new relationships are essential
  - "extend" boxes with new ports, new capabilities

- *New*: optimizations are essential
  - break abstraction boundaries to achieve *efficiency* or *availability*
  - how non-functional "features" or "qualities" are added by transformations



With These Missing Pieces

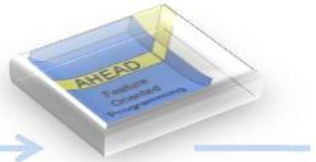


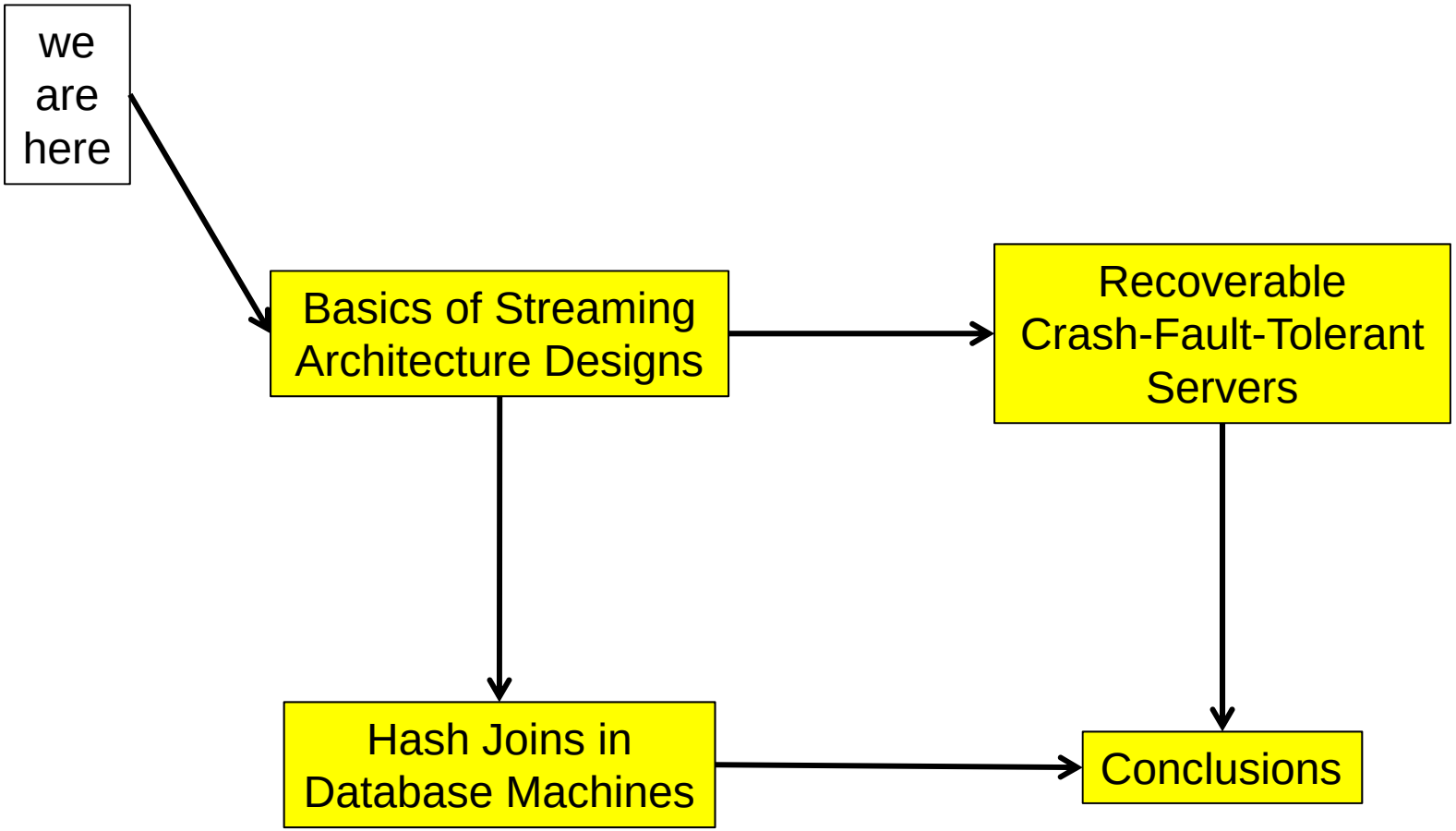
- Show how complex architectures designed *by experts* can be:
  - explained ***incrementally***
  - verified ***incrementally***
  - built ***incrementally***
  - tested ***incrementally***

  by *non-experts* from first principles – 2 very different domains
  recoverable crash-fault-tolerant servers and
  join parallelization in database machines

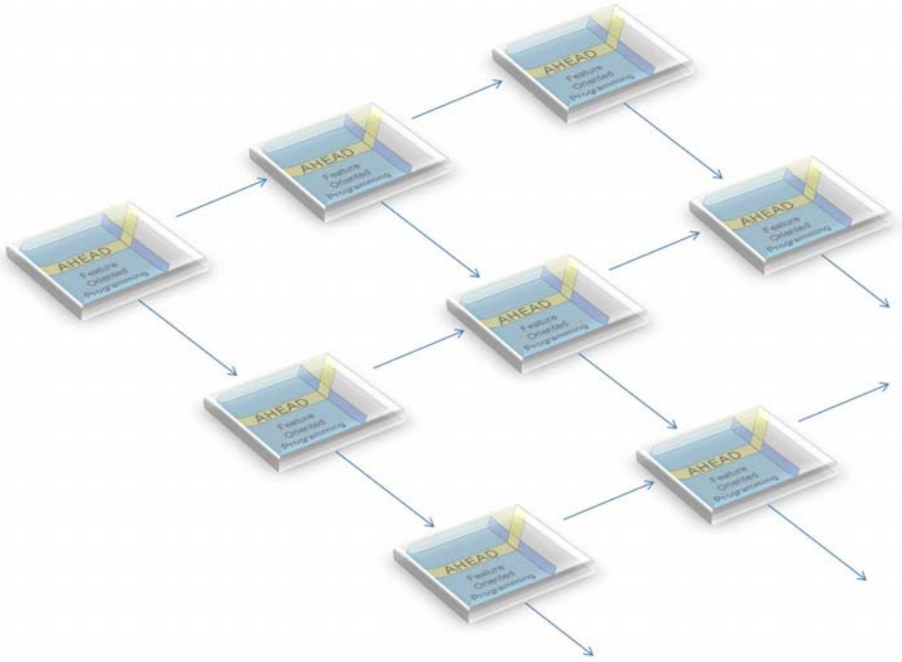
- Not just “cute” (nice-to-have)
- Essential to accomplish the above
  - experts create these designs with implicit knowledge
  - by making principles explicit, non-experts like you and me can participate and appreciate the challenge that experts face & techniques they use

Series of Mini-Talks

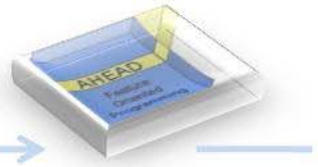




#1: Basics of Streaming Architectures



Basics



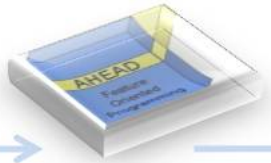
- **Component-Connector Architecture** is a directed graph
  - box — component or computation
  - arrow — communication path for messages, tuples drawn in direction of data flow
- Semantics of box is clear from context

1, 50, 2, 62, 53

1, 2, 50, 53, 62

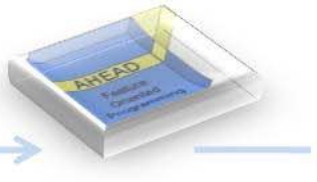
- Elide unnecessary details (sort key, sort order, sort type)

Hierarchical Refinement

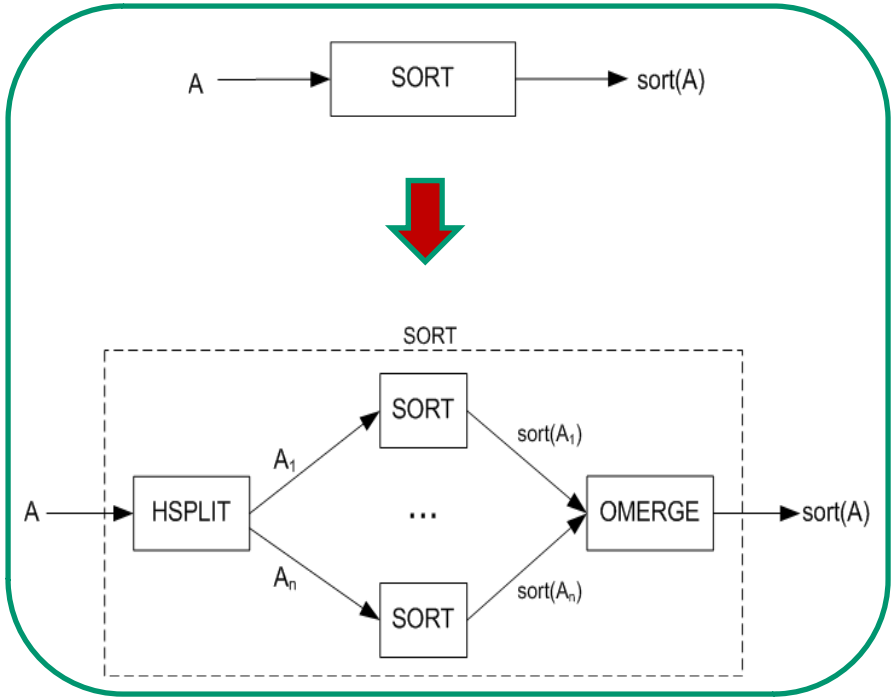


parallelizing refinement

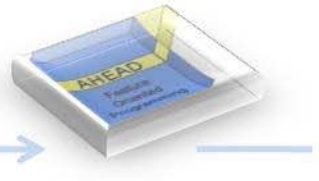
Transformations



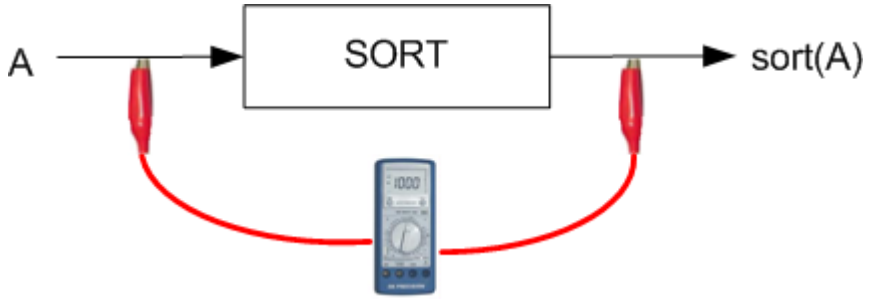
- Refinement is a **transformation**
  - input pattern → output pattern
- All **transformations** we use can be proven correct
  - algebraic identity
  - simple enough that intuition suffices
- **Correct by Construction**



Testing



- Most details are not captured in architecture diagrams
  - whether SORT box works correctly
  - typically boxes don't have proofs of correctness – need to test

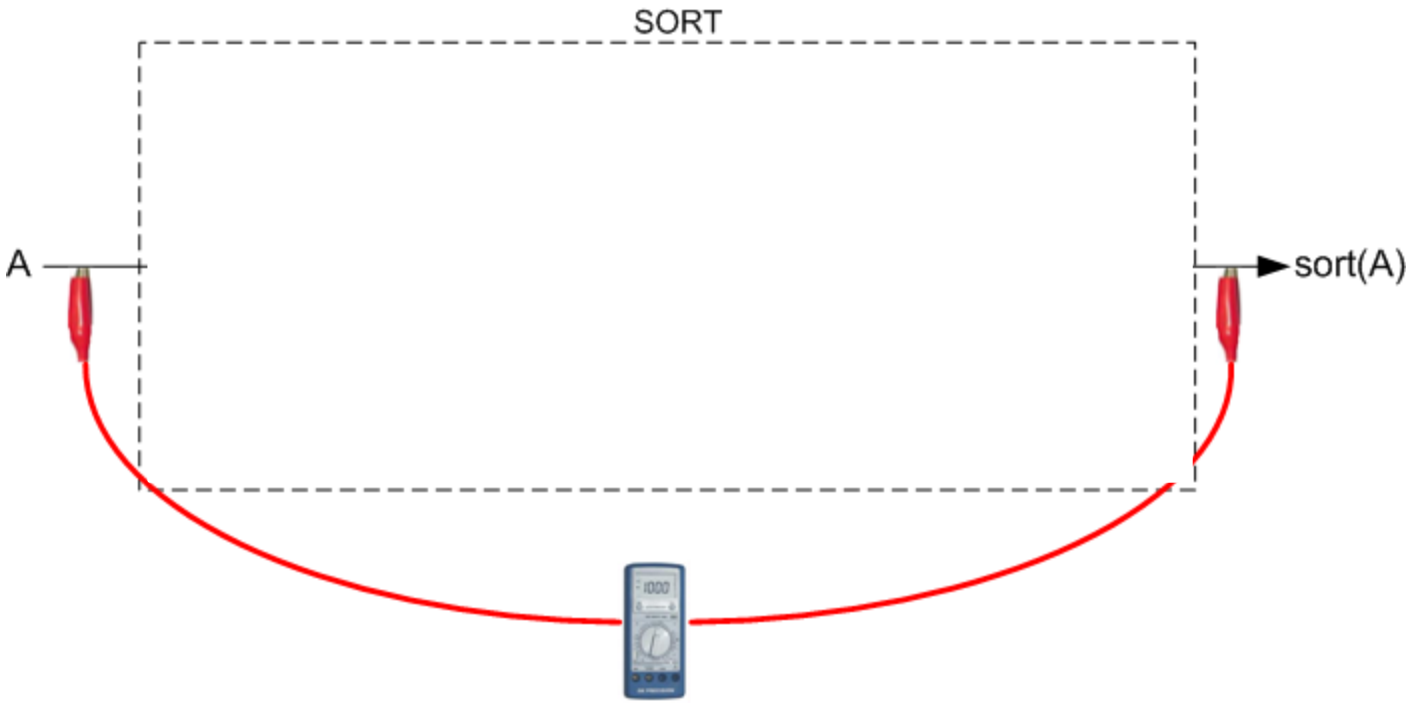


"*Beware of bugs in the above code; I have only proved it correct, not tried it.*"
– Donald Knuth

Module (Unit) Test

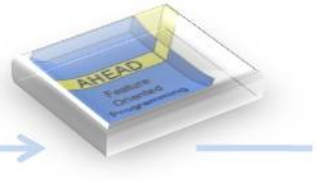
Testing Refinements

We identify tests at every level of abstraction.
The validity of these tests (properties) must hold after every refinement.
So as details are revealed, we accumulate tests to
verify the correctness of our implementation.



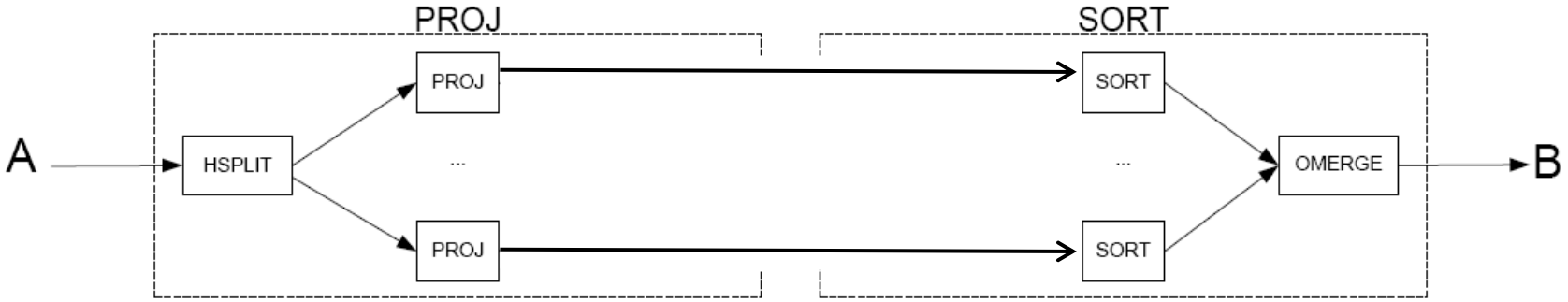
Integration or System Test

Key: Optimizations

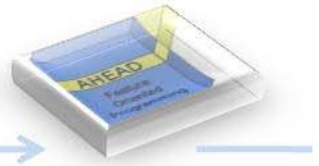


- Break encapsulations to achieve non-functional properties
  - efficiency or availability
  - applying algebraic identities that do not change design semantics

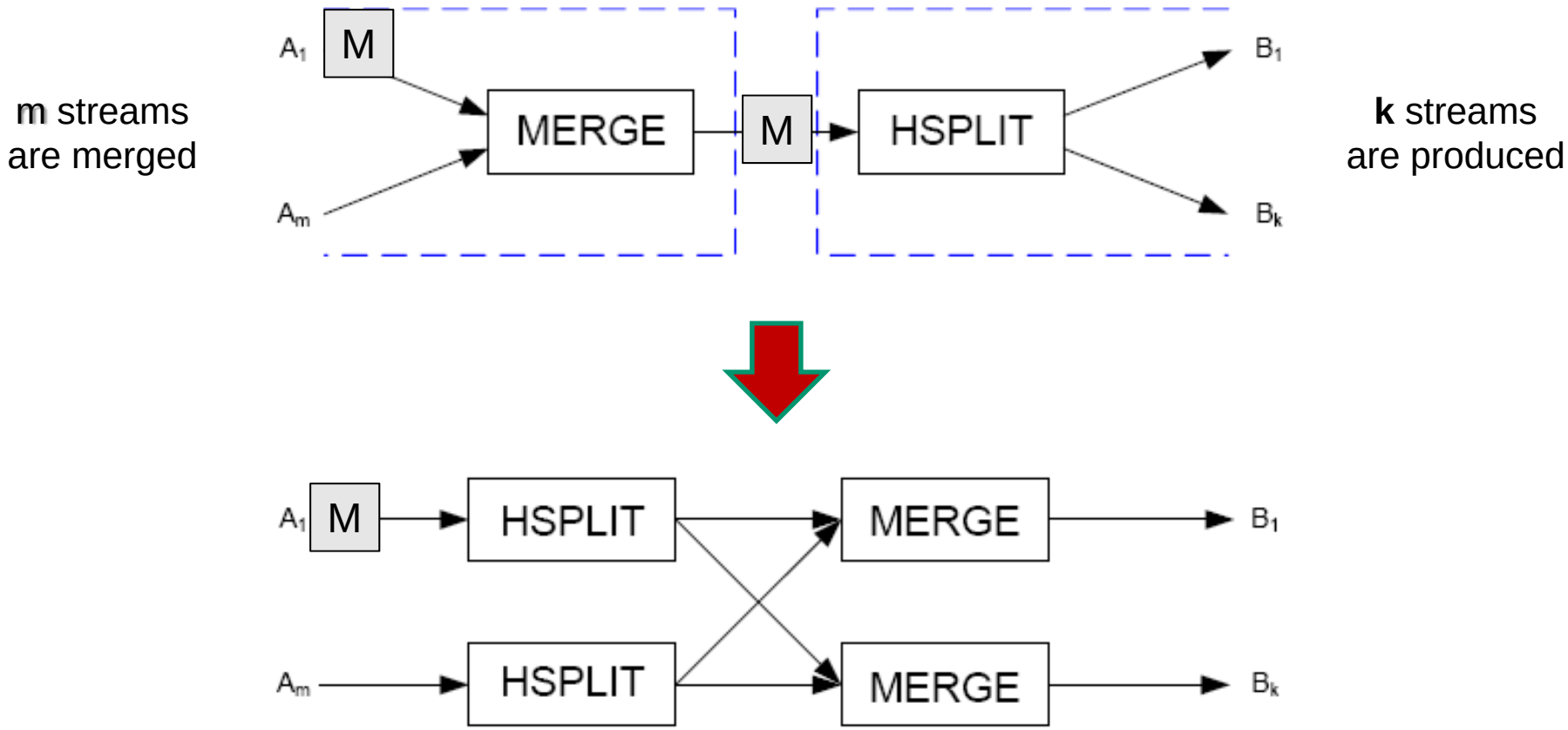
Same HSPLIT box (hash same attributes, same hash function)



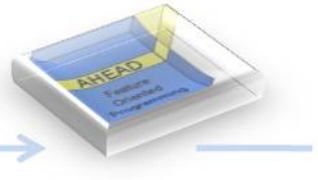
Exchanges



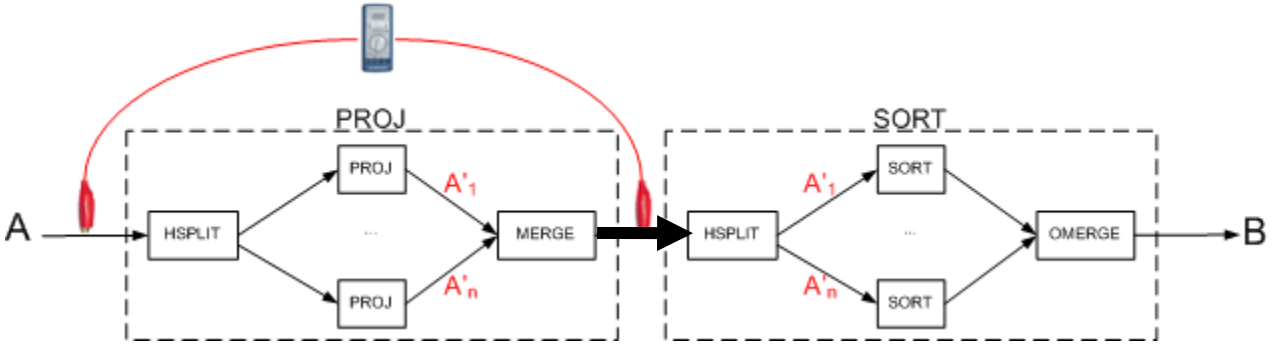
- Optimizations that reorder *stateless* computations
  - ex: property that each A_i message is assigned to a single B_j stream

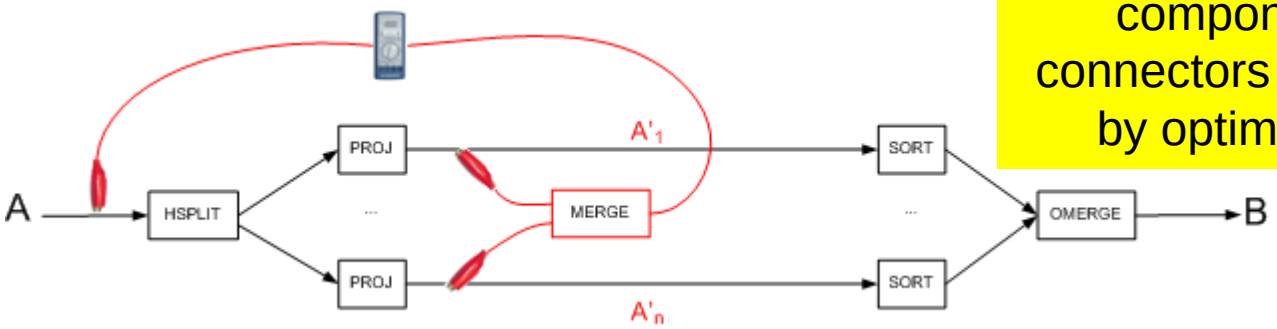


Optimizations and Tests



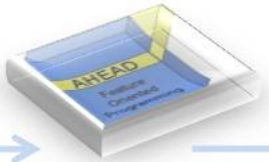
- Optimizations (exchanges) play havoc with tests
- Tests must be transformed



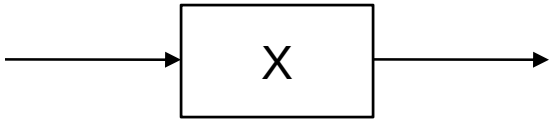


Rule: testing may reintroduce components & connectors eliminated by optimizations

Lastly, Extensions

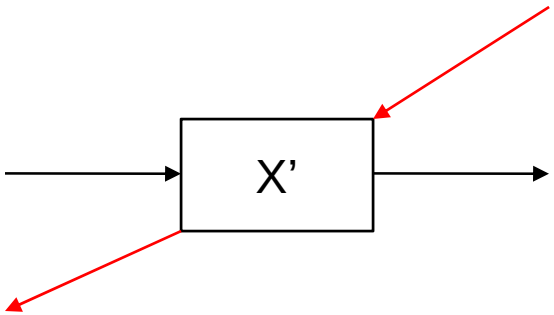


- A transformation can extend the capabilities of a box and add new connectors

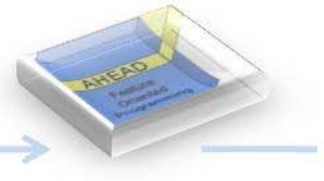


- Extensions add “features”

- Accomplished by preprocessors #ifdef inclusion of extra code
- Or by more sophisticated means

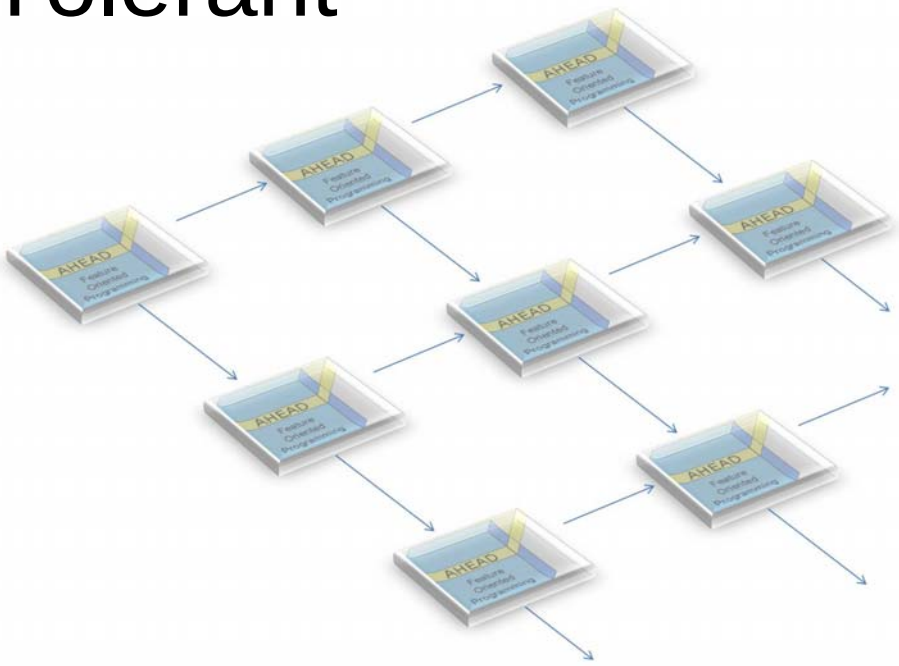


Recap

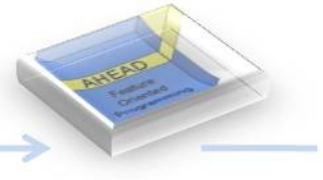


- Component-connector architecture is implementation model
  - transformations progressively elaborate models
  - each is a refinement or optimization
  - each applies a ***behavioral substitution*** (Liskov)
- Every transformation is simple and can be proven correct
- Use tests to verify implementations
- Now, let look at some real examples…

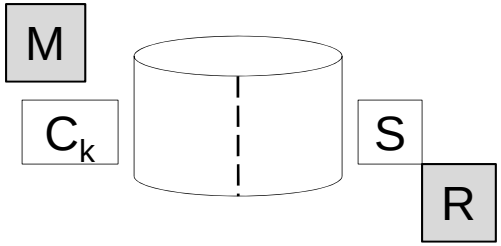
#2: Recoverable Crash-Fault-Tolerant Servers



Overview

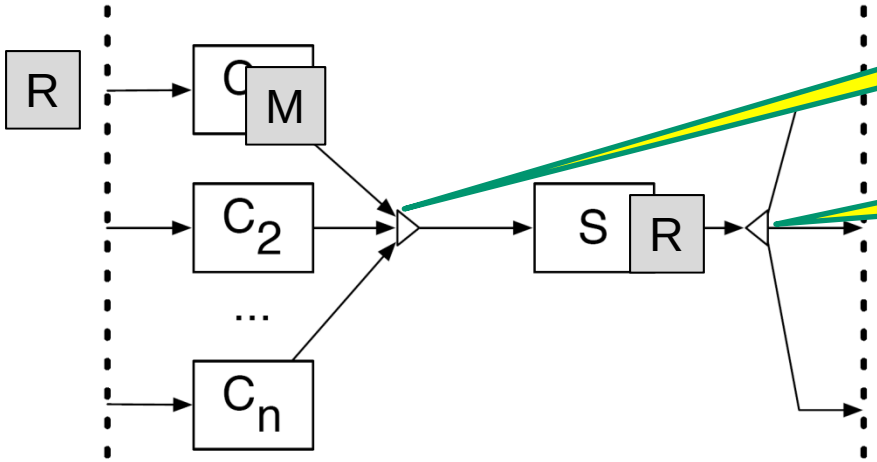


- Sequential server architecture has a cylinder topology



assume server updates state

- Unroll cylinder by breaking along the seam

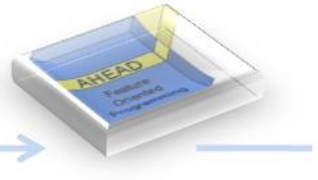


serializer

demultiplexor

broadcast

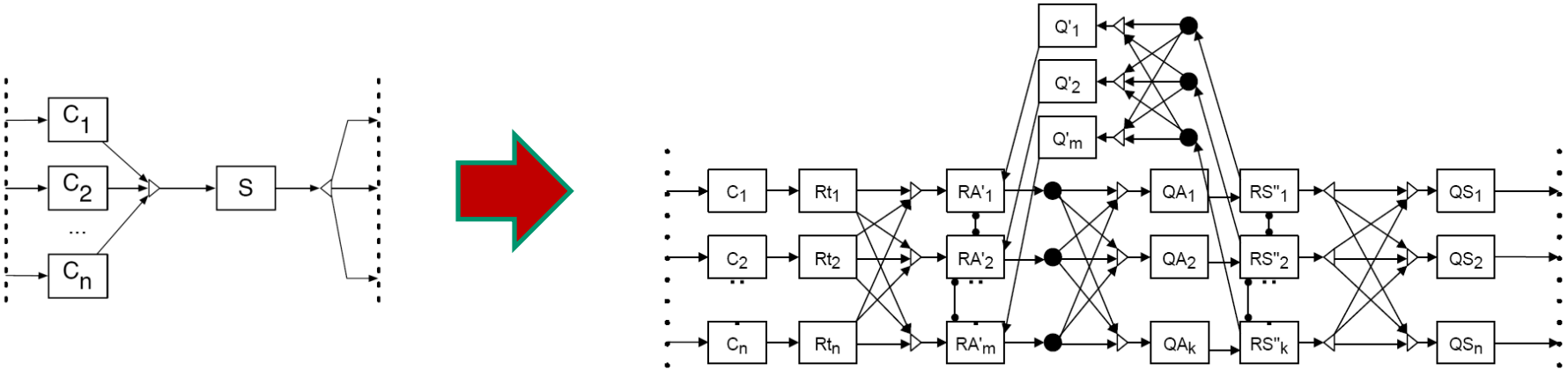
Our Goal



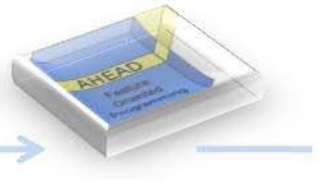
- Transform a vanilla sequential server architecture to a

  Recoverable, Crash-Fault-Tolerant Server

  - consider CFT transformations first
  - recovery transformations last



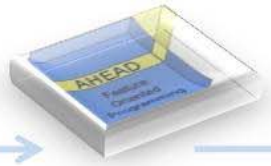
Crash-Fault-Tolerance



- Ability of a service to survive a number of failures

- **Failure** – when a box stops processing messages
  - no messages pass through a failed box
  - a failed box cannot create new messages
  - *assume each box executes on its own machine*
    - but multiple boxes can run on single machine
    - if machine fails, all boxes on that machine fail
    - failures do not propagate across machine boundaries

Standard Failure Assumptions



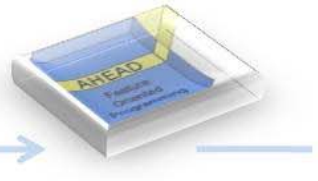
- Failures of network components

  serializer (◄)
  demultiplexor (►)
  broadcast (●)

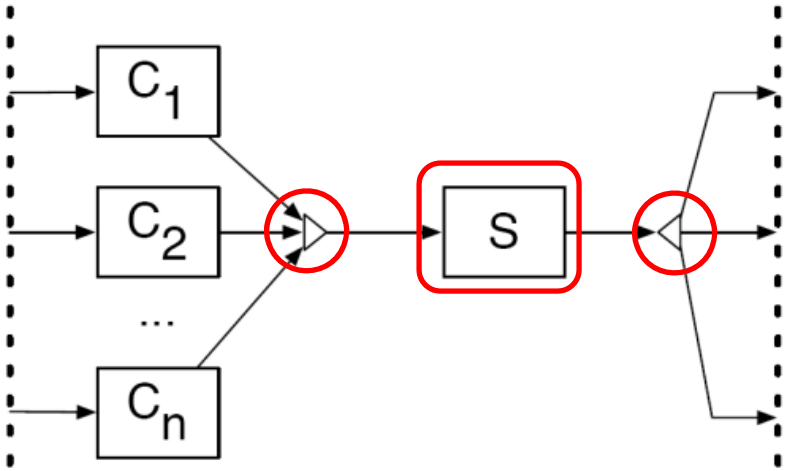
  affect a machine the same as pure software boxes
  - ex: a machine can't process requests if its network card stops working

- Do not depend on synchronous networks
  - do expect eventual synchrony
  - use **retransmissions** (in application, network protocol, or both) to deal with transient packet loss

Technical Goal of CFT

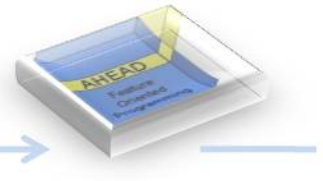


- Eliminate **Single Points of Failure (SPoF)**
  - a failure of a single box (machine) causing the server abstraction to fail
  - our current design has 3 SPoFs

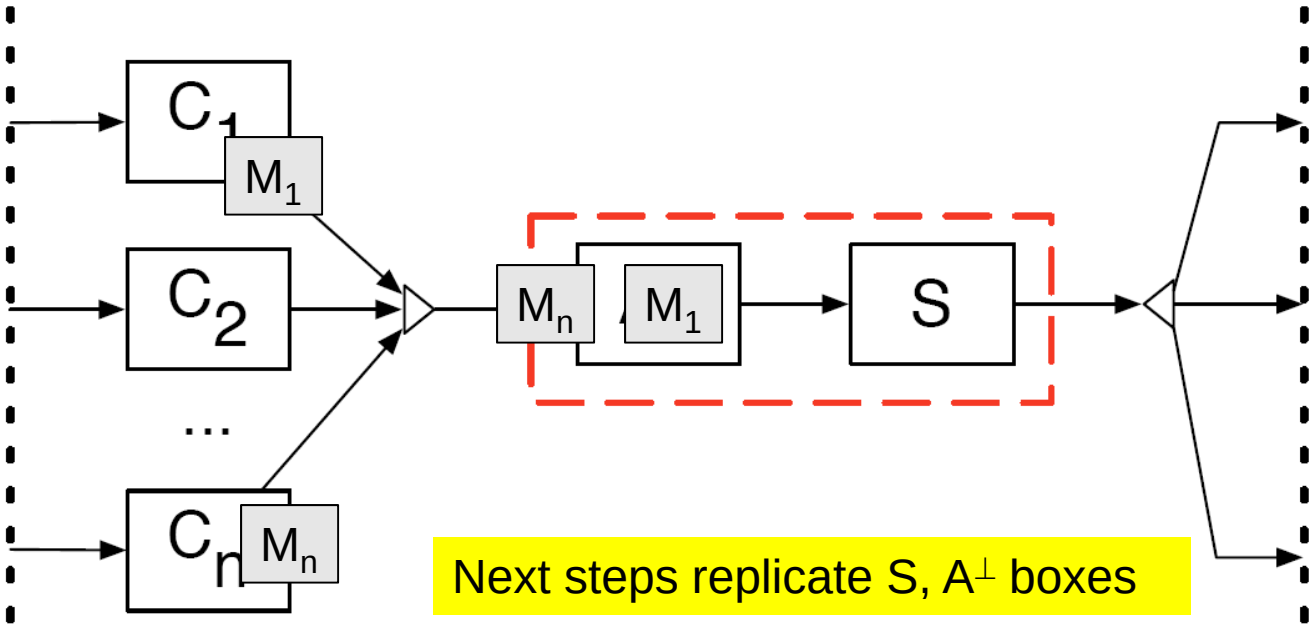


- "Solve" problem by replicating boxes
  - not only solution – we follow most advanced solution to date
  - to appear SOSP 2009

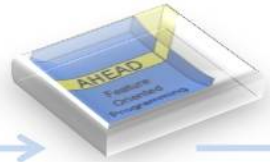
Step 1: Agreement



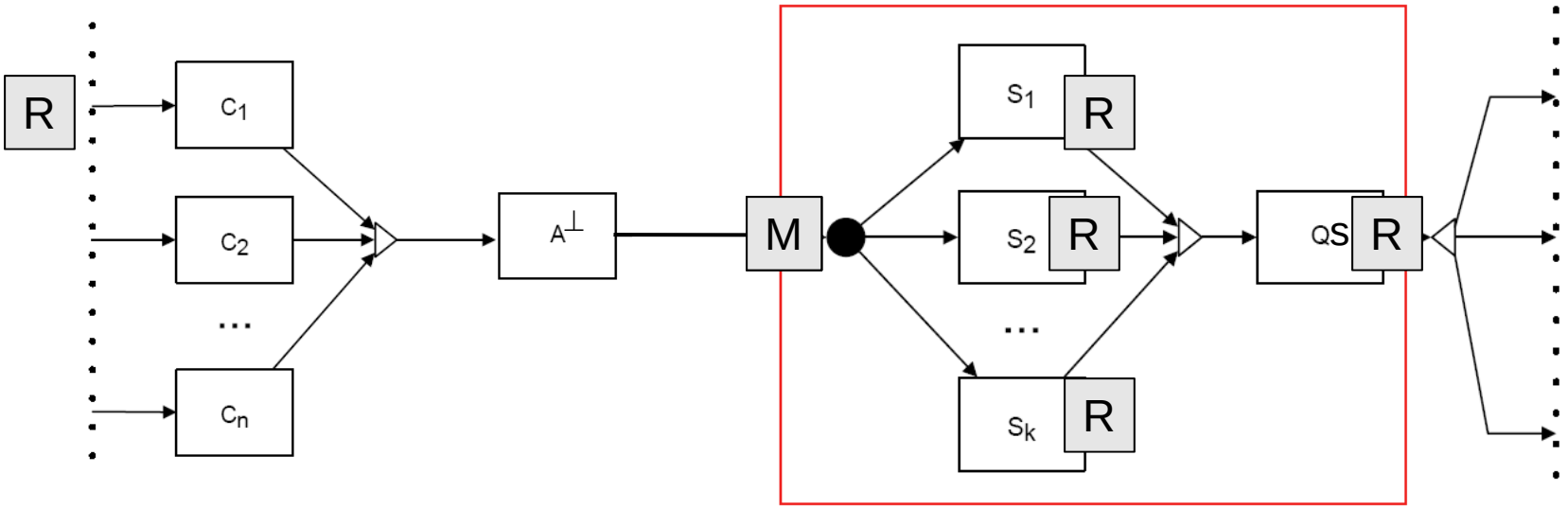
- Add an agreement node $A^{\perp}$
- $A^{\perp}$ is an ordered queue of messages, passing messages one at a time to the server
- In effect, $A^{\perp}$ does nothing it is a place holder for later refinements



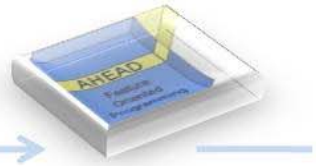
Step 2a: Replicate Servers



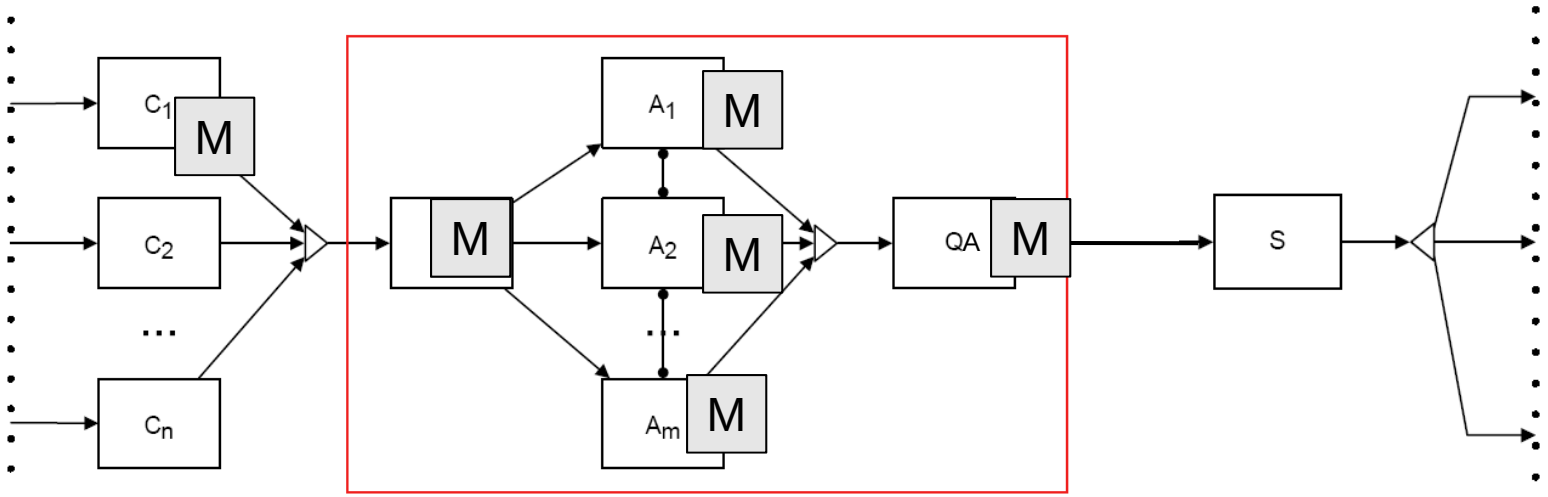
- Make k copies of server
- Each server receives exactly the same sequence of messages from the $A^{\perp}$ abstraction
- QS collects a quorum of identical messages;
  transmits message when a sufficient number of copies are received
- *Refinement emulates abstraction of a single correct server*



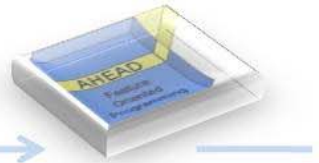
Step2b: Replicate $A^{\perp}$ Boxes



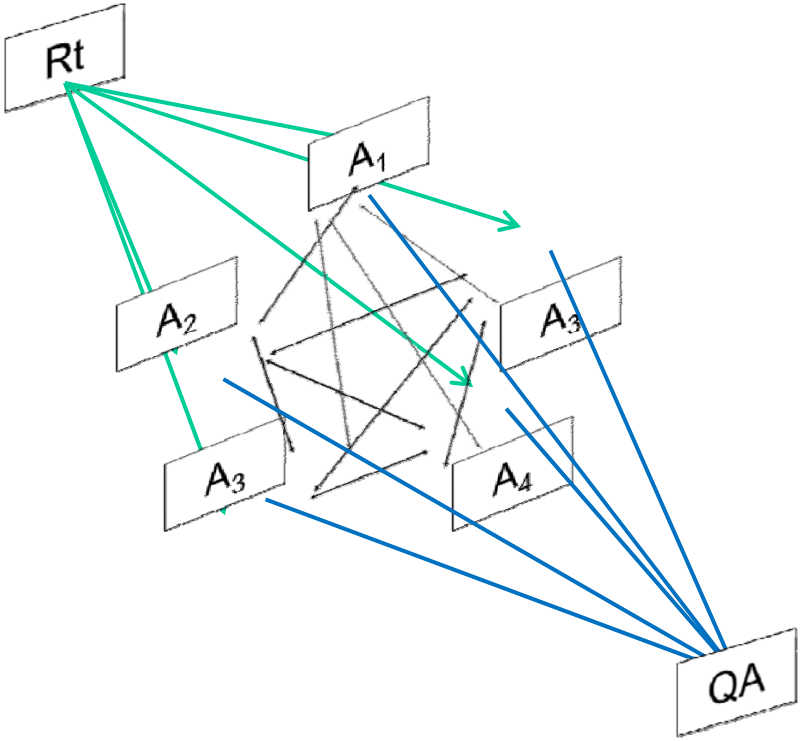
- Make m copies of $A^{\perp}$
- Client requests are routed by box Rt to *some or all* A replicas
- A replicas run an **agreement protocol** (Lamport 1998) to decide which is the next client message to process
- A replicas vote and a quorum is taken by QA; when a sufficient number of identical messages is received, QA forwards the message
- *Refinement emulates abstraction of a single queue*



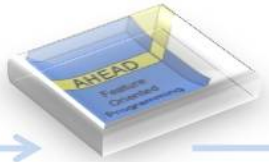
Pause…



- The A replicas talk to each other; topology is conical



Why are A Replicas Needed?



- Ans#1: tolerate failure of multiple A boxes

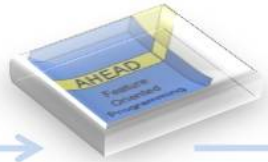
  tolerate *f* failures, need *m* = *2f+1* replicas of A

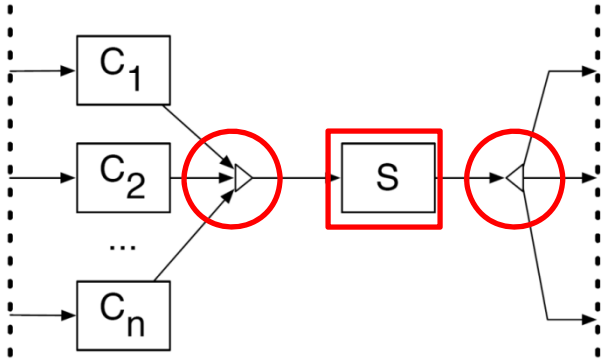
- Ans#2: a consistent order of requests is essential for correct server behavior

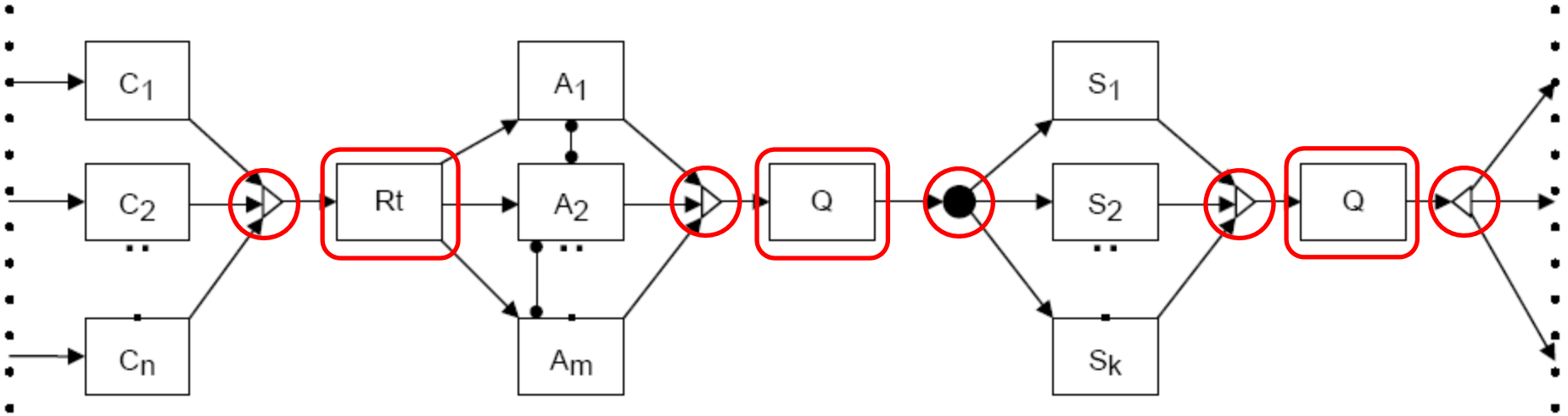
- If S replicas processed client requests in different orders, server replica states would diverge and responses from different servers for a single client request would be inconsistent

- Inconsistencies violate the one-correct-server abstraction

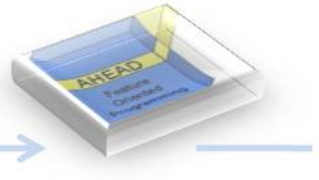
Where We Are…





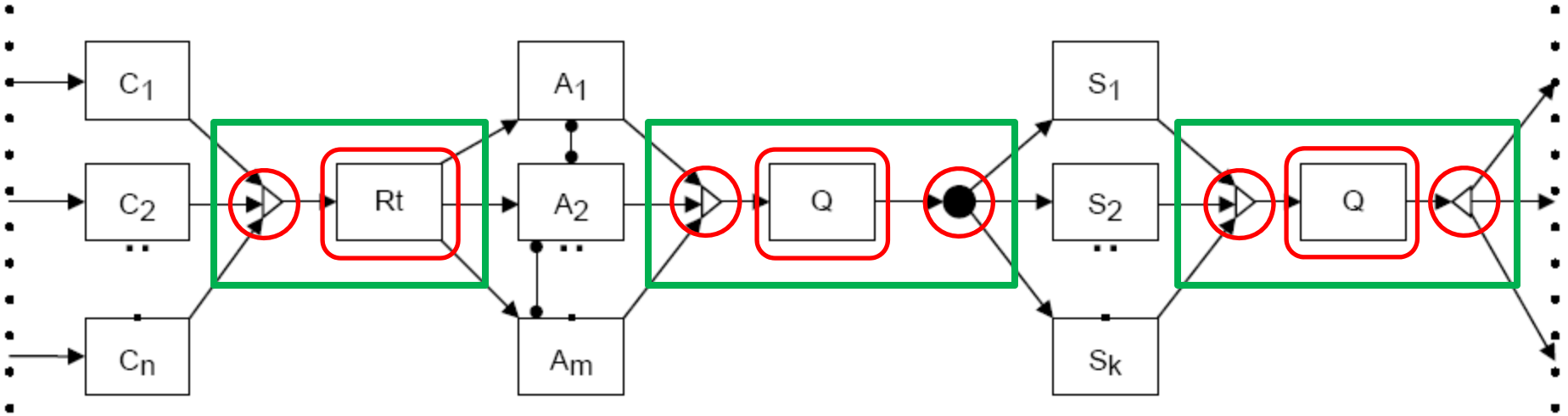


Where We Are Going

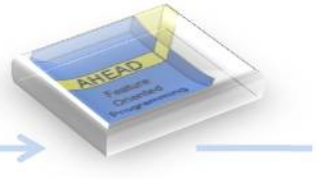


- Apply optimizations (exchanges) to eliminate SPoFs

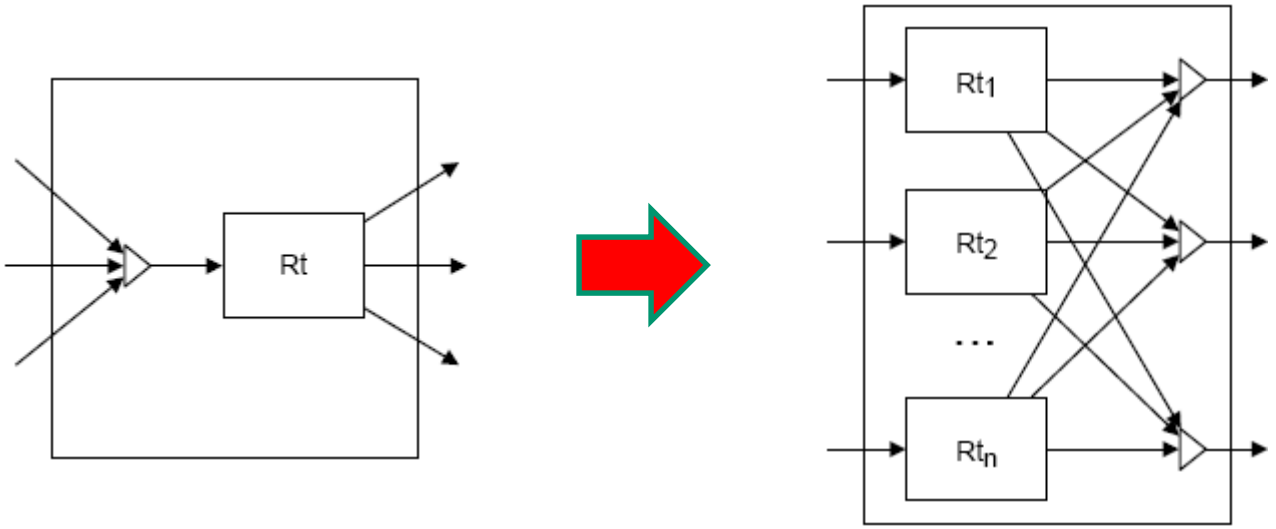
- Define new (green) abstractions, and replace them with implementations that have no SPoFs

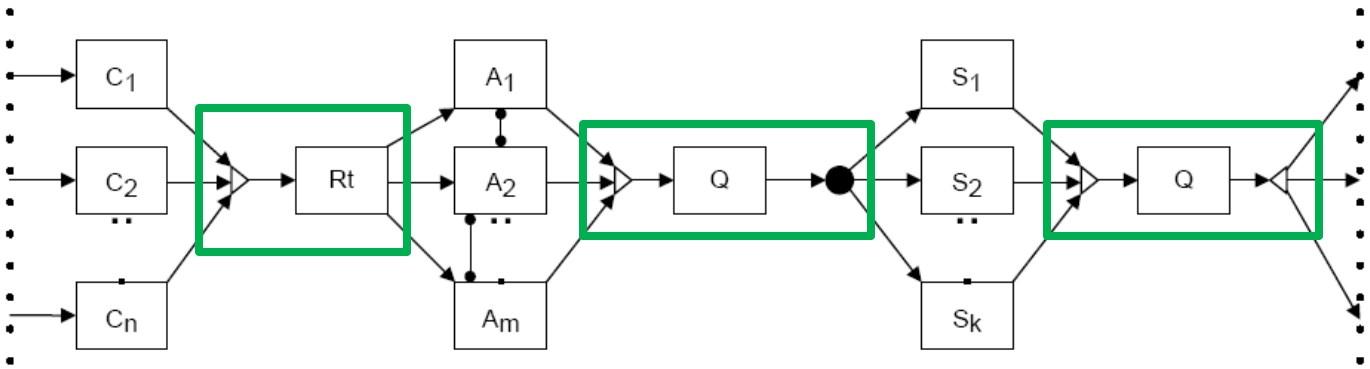


Step 3a: (►,Rt) Exchange

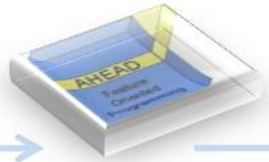


- Each client request is sent to a subset of A replicas

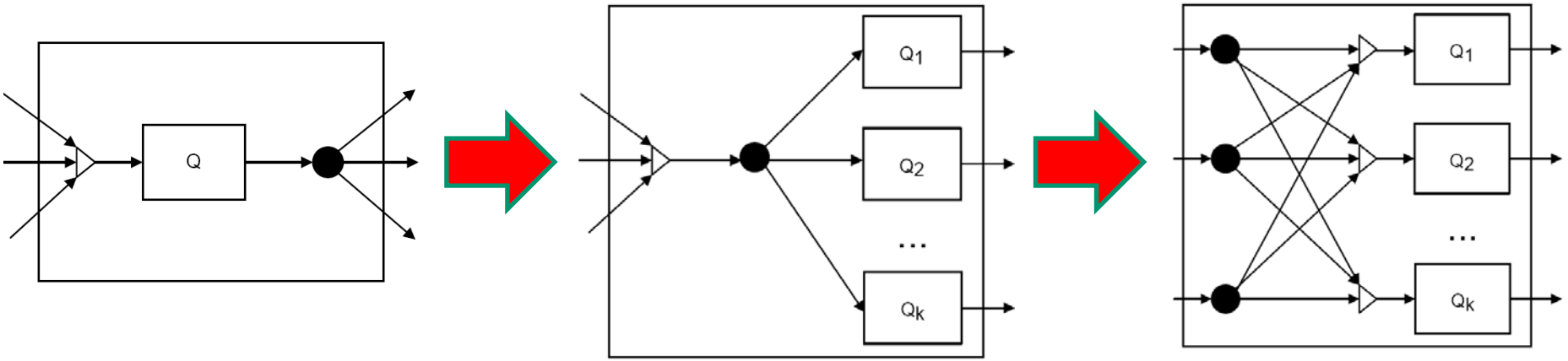


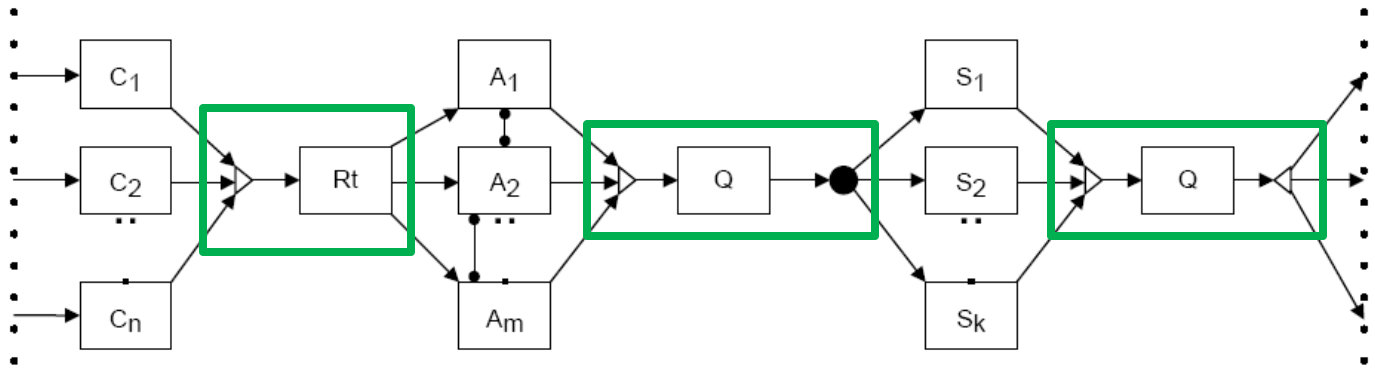


Step 3b: (▶, Q, •) Exchange

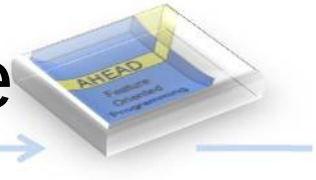


- Each quorum-decided request from replicated A boxes is delivered to all Server replicas

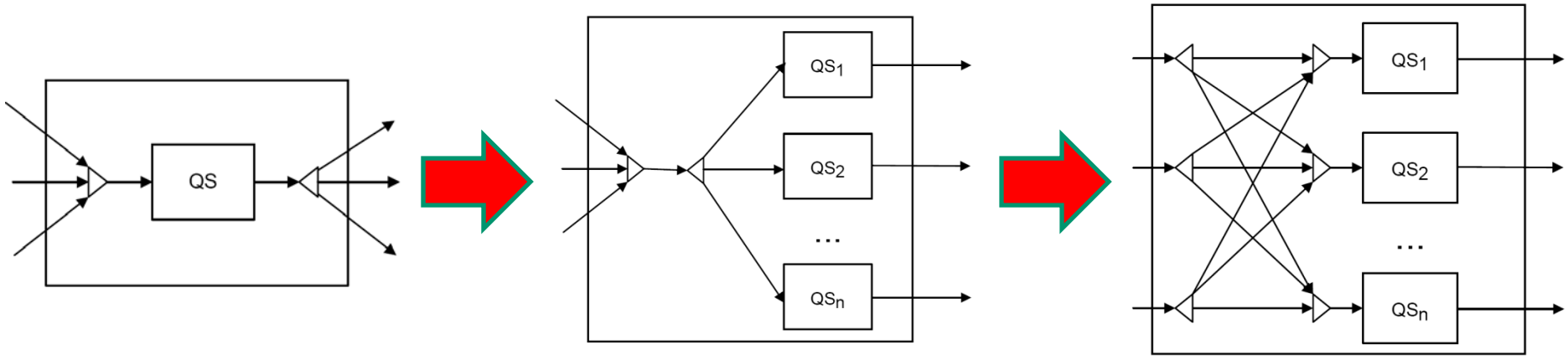


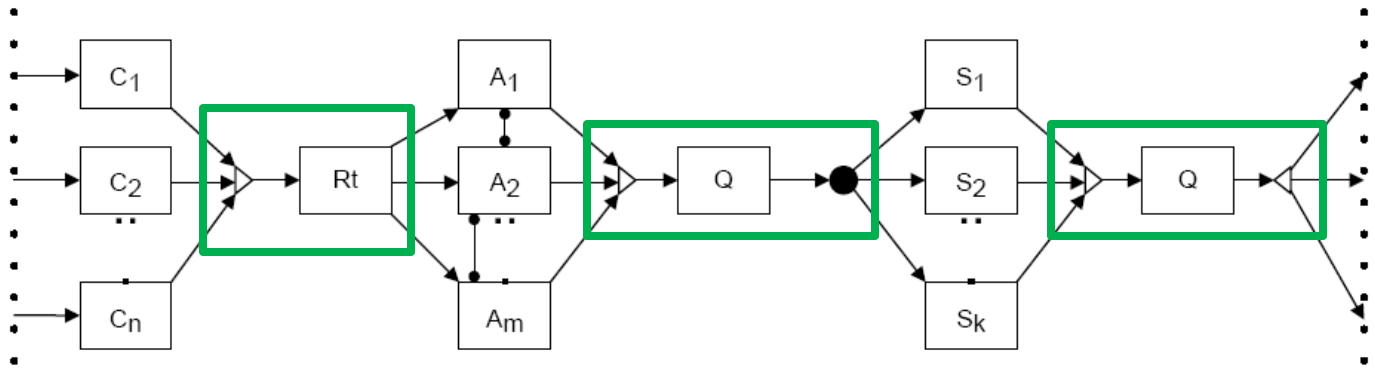


Step 3c: (►, QS, ◄) Exchange

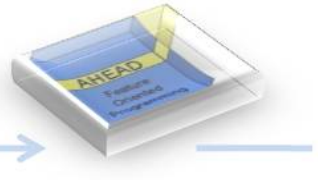


- Each quorum-decided response from replicated S boxes is received by a client box

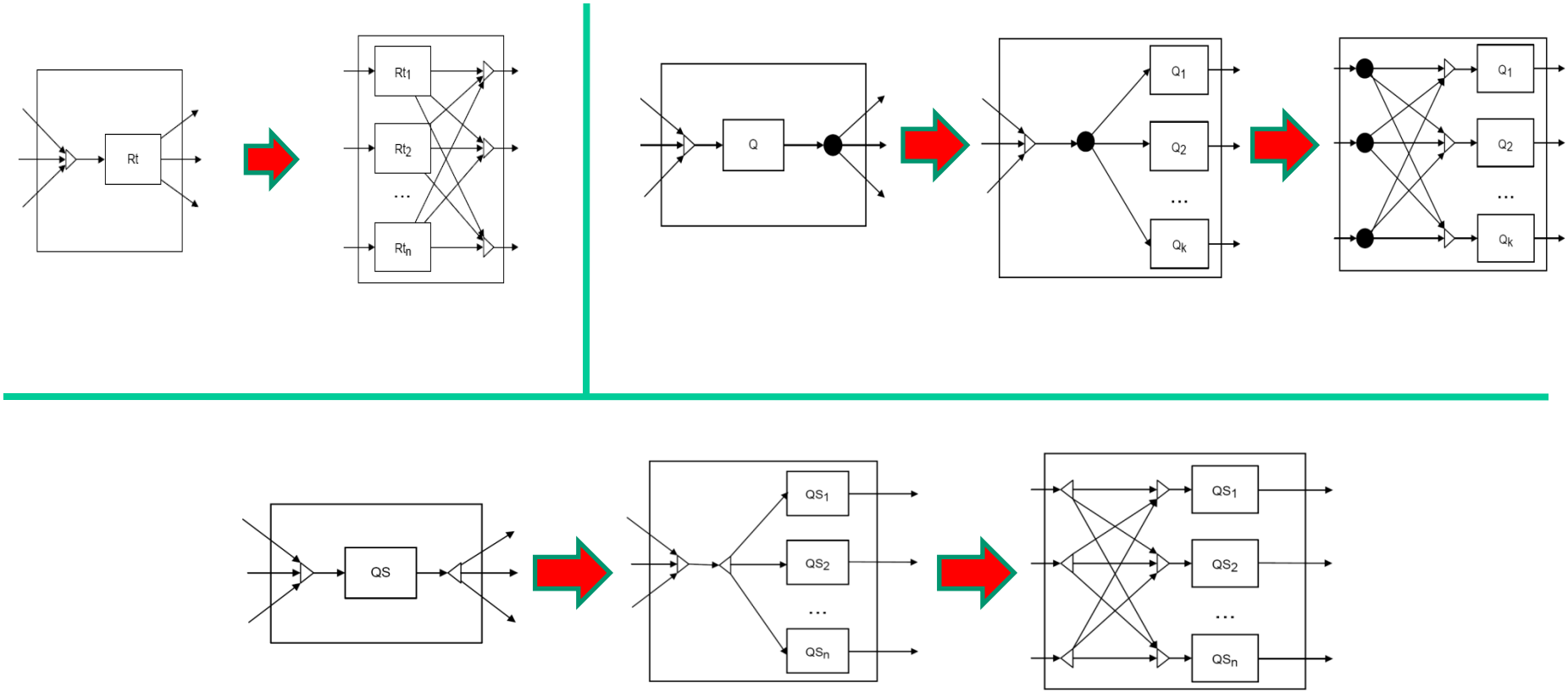




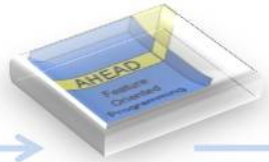
Why So Simple?

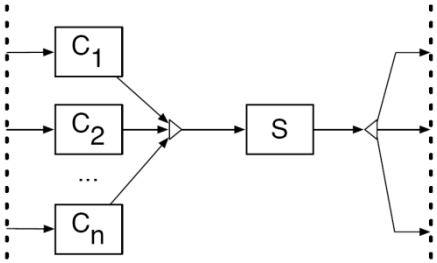


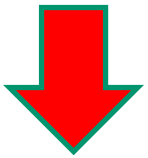
- All exchanges involve stateless computations
  - state would require a heavy-weight solution (agreements, etc.)



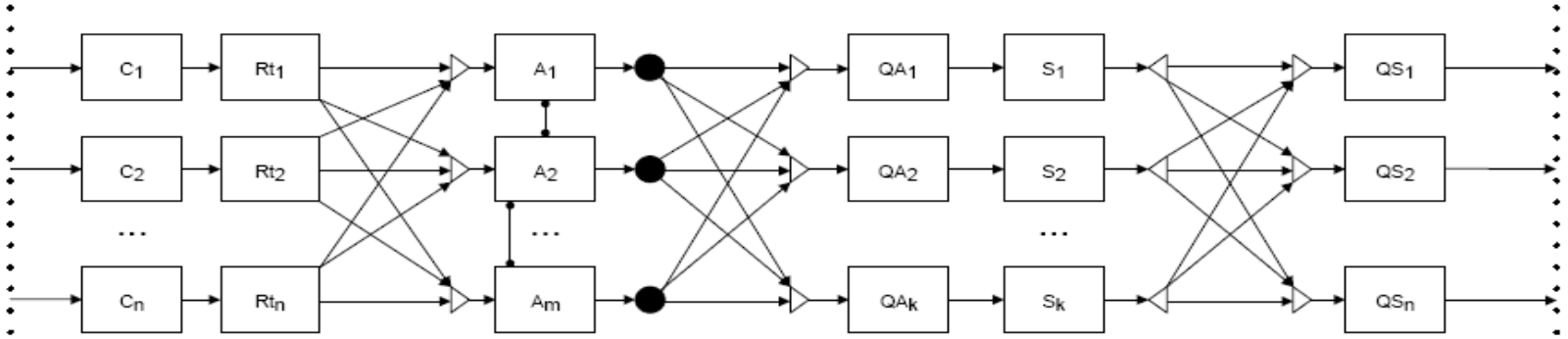
Final CFT Result



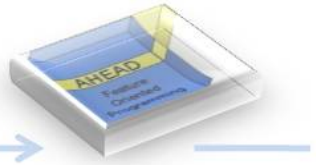


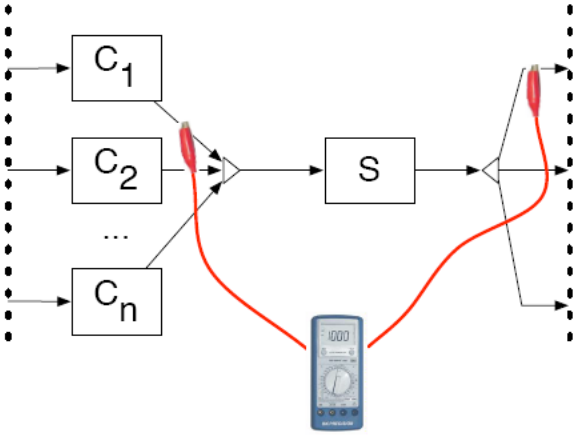


No SPoF



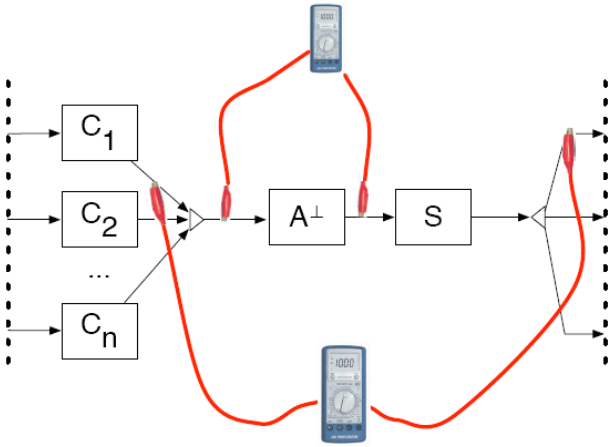
Testing CFT Abstractions



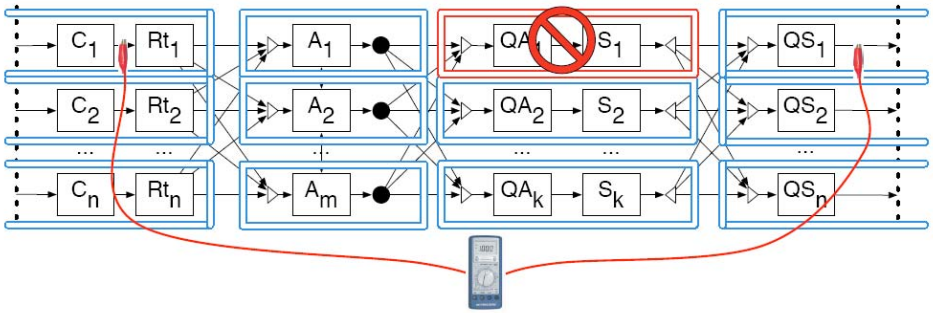


Test 1

Test 2

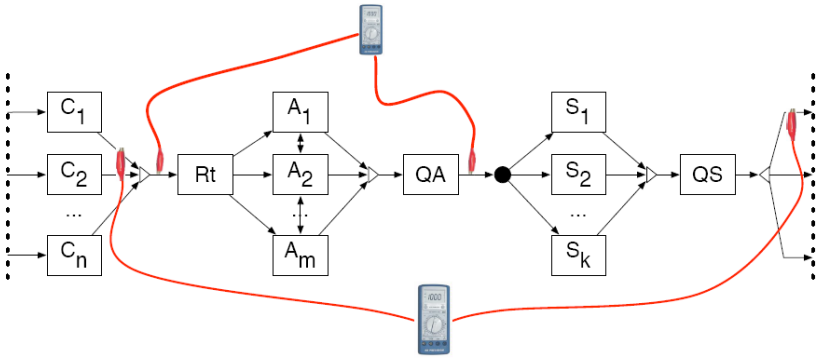


Test 1

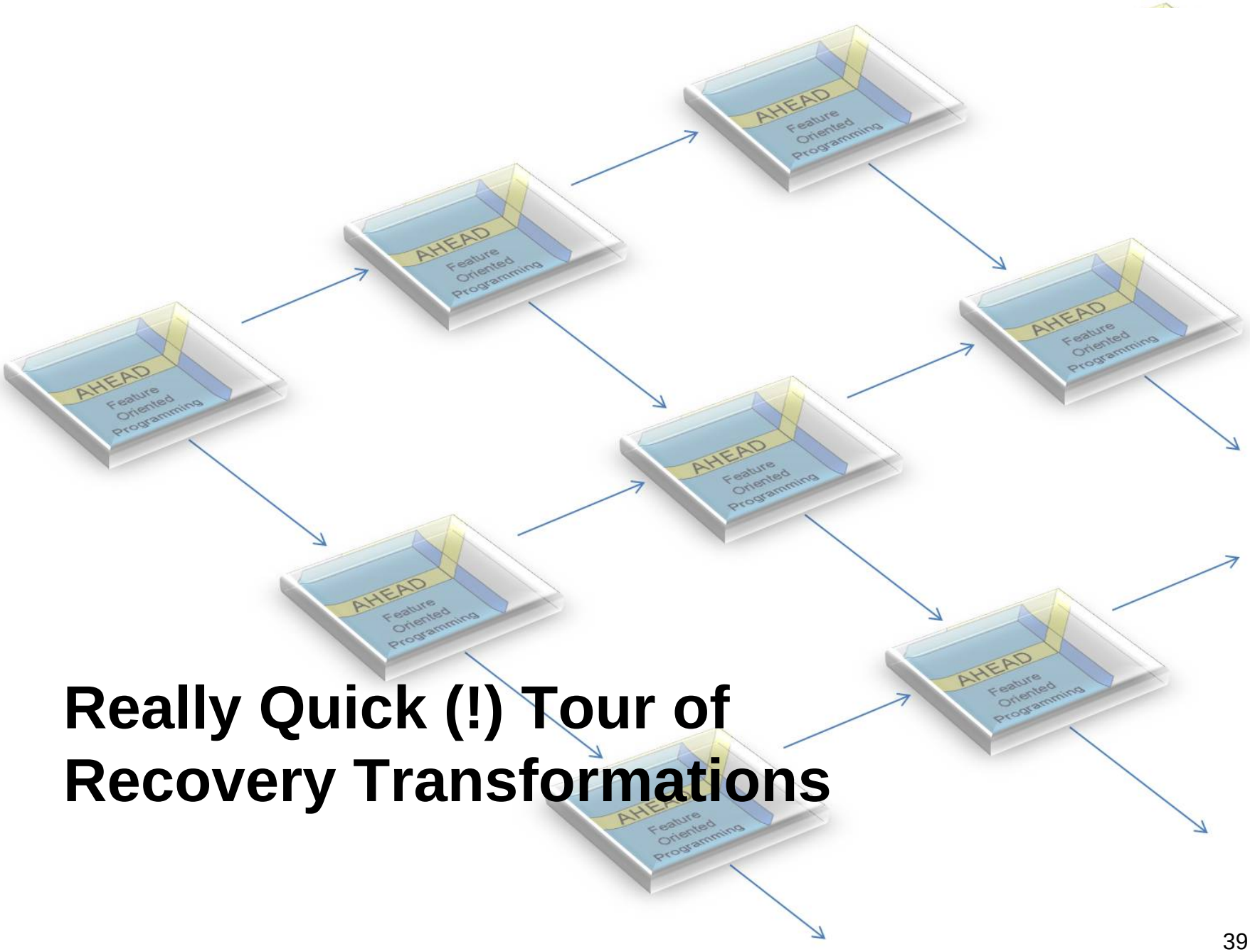


Test1

Test 2

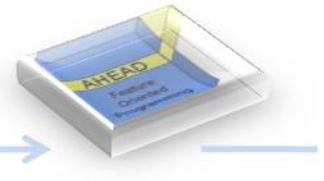


Test 1



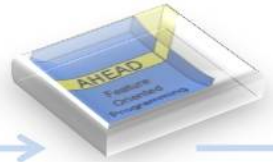
Really Quick (!) Tour of Recovery Transformations

Overview

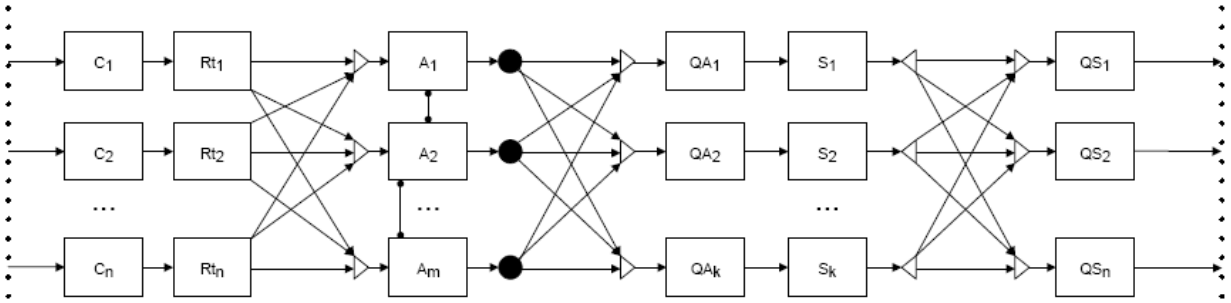


- Just as databases can recover from machine failures, so too can servers
- Recovery limits the situations where clients see unresponsive server abstraction
- Recovery is added by a series of transformations, not unlike the transformations to add CFT

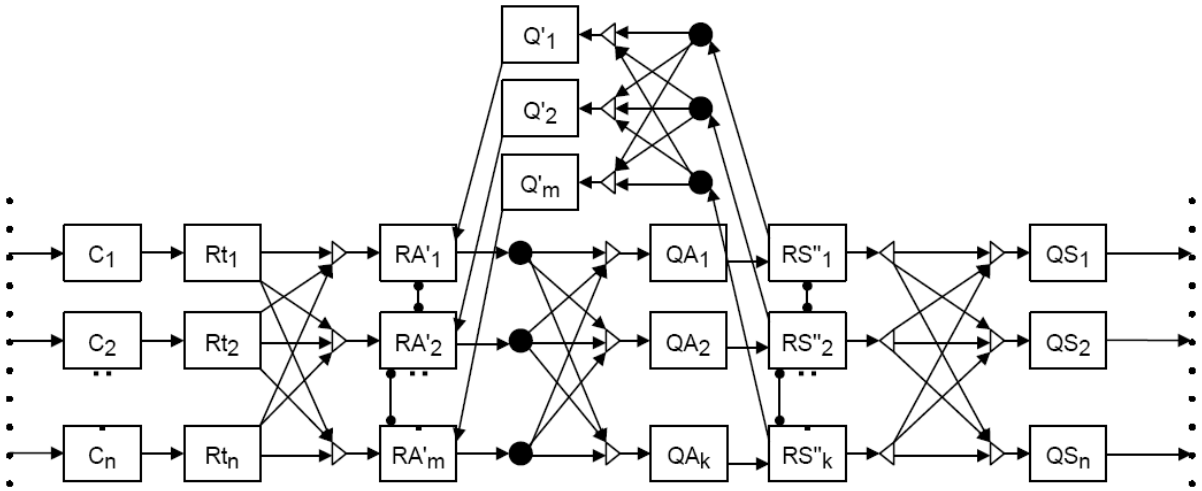
Recovery Transformation



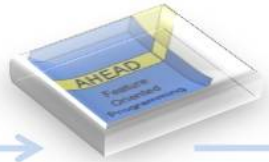
CFT
Server
Architecture



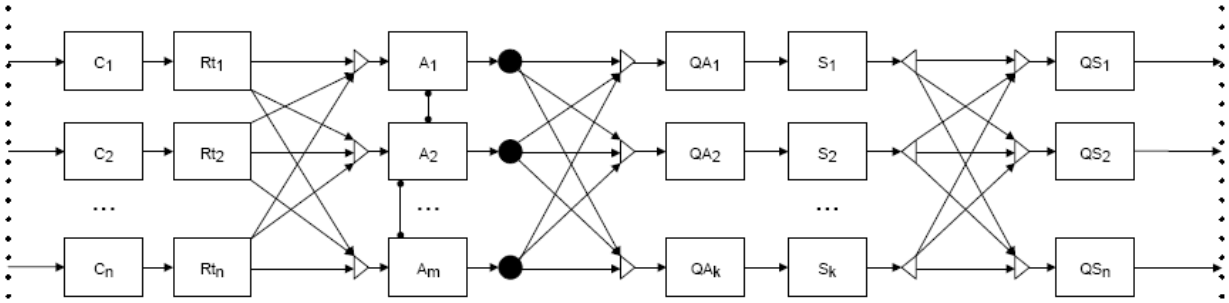
Recoverable
CFT
Server
Architecture



Changes: New Relationships

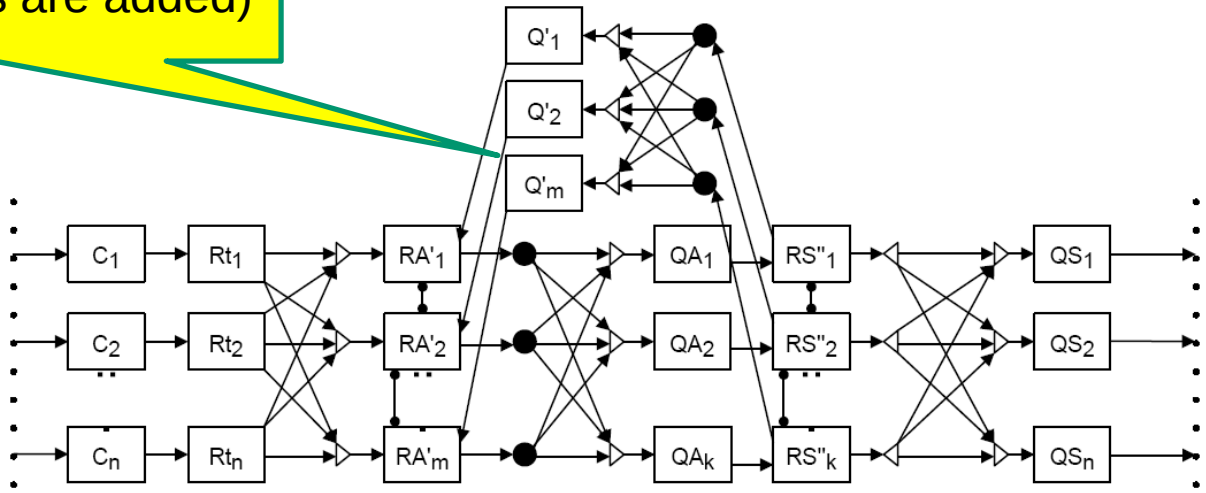


CFT
Server
Architecture

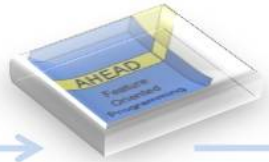


Servers now talk to A boxes
(new relationships are added)

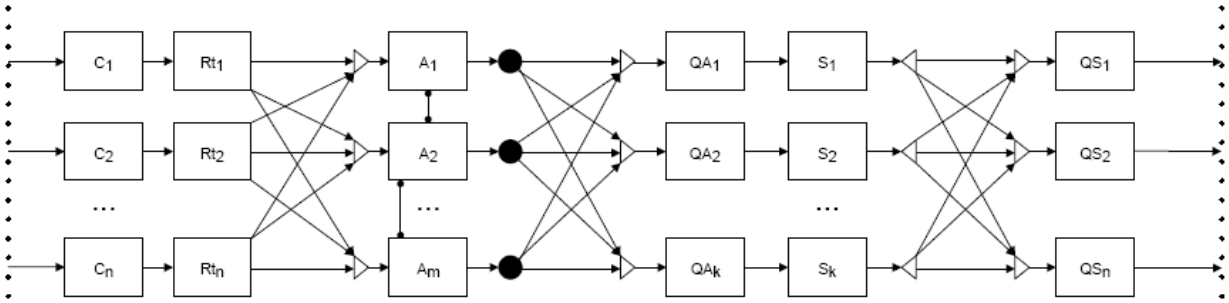
Recoverable
CFT
Server
Architecture



Changes: New Relationships



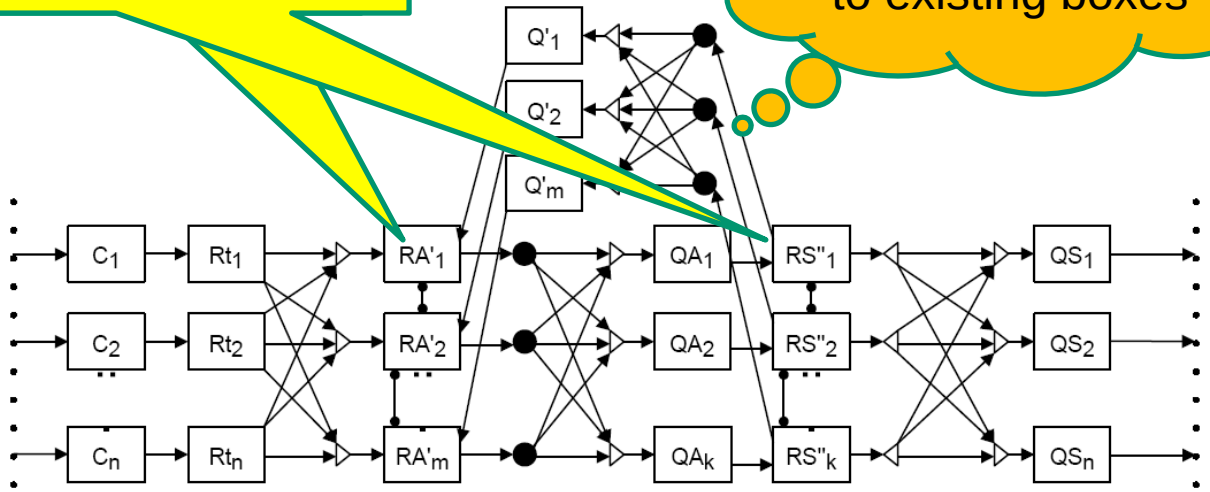
CFT
Server
Architecture



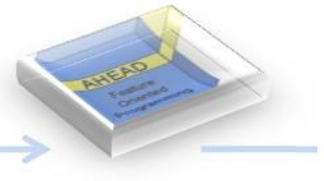
A and S boxes are extended
(new ports, capabilities added)

incrementally
adding features
to existing boxes

Recoverable
CFT
Server
Architecture



Quick Recap

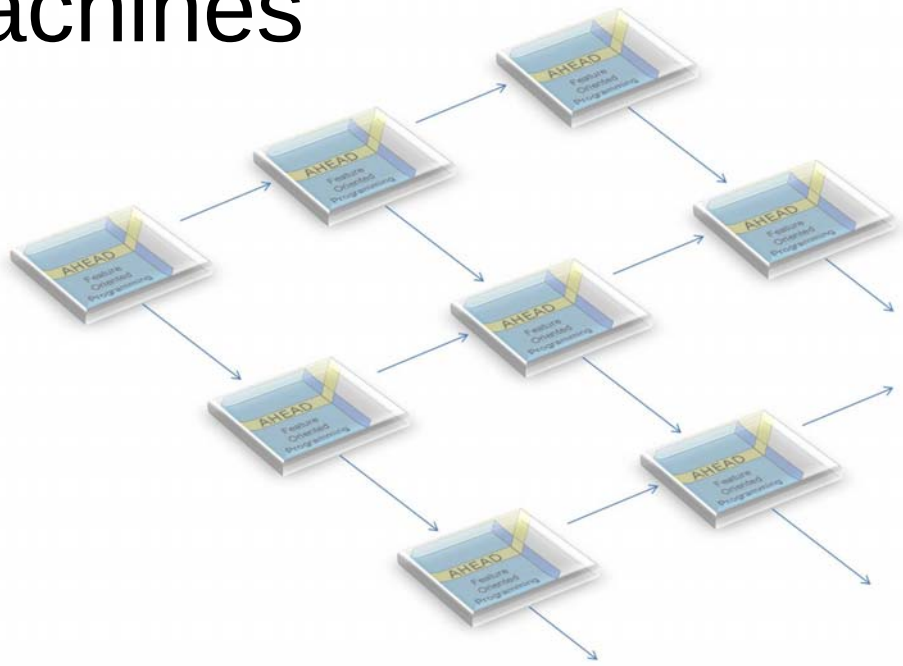


- Incrementally explained design created by experts to map a vanilla server to a recoverable, crash-fault-tolerant server
  - withstand failures
  - we are using component-connectors to explain, automate their designs

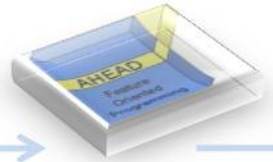
- *Still a lot of engineering left to do*; but our design provides guidance on how to build, test these systems incrementally

- Now look at a very different domain where a sequential architecture is mapped to a parallel architecture using *exactly the same principles*

#3: Parallelizing Hash Joins in Database Machines



Gamma Database Machine



- Gamma was (maybe still is) the most sophisticated relational database machine ever built in academics
    - University of Wisconsin late 1980's early 1990s
- Look at how hash joins were parallelized
    - fundamental result in parallelizing joins
    - representative of commercial systems today
    - presented in a new way
    - derive Gamma hash join architecture from first principles

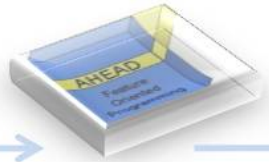
Sequential Hash Join Architecture

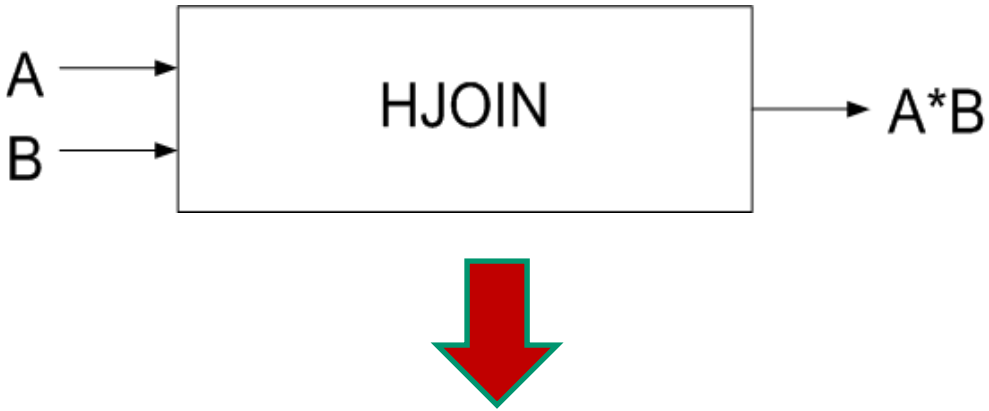
- Hash join takes 2 streams of tuples (A,B) as input and produces the join of these streams (A*B)

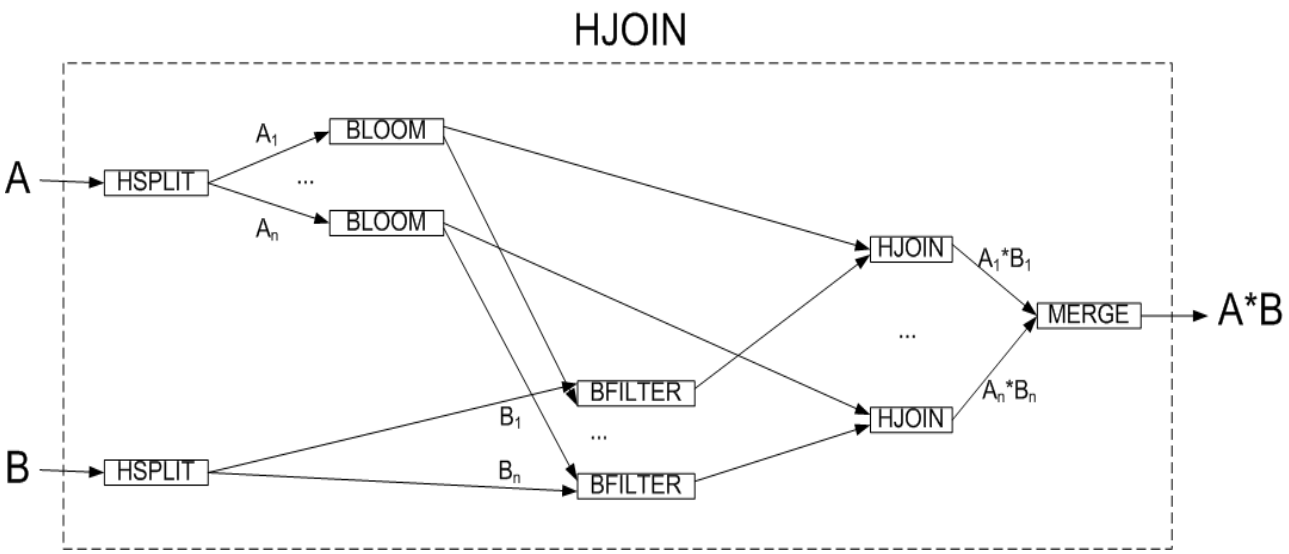
- Algorithm:
  - read all of stream A into memory in a hash table
  - read B stream one record at a time;
    hash B's record and join it to all A record's with the same key
  - linear algorithm

- *How did Gamma's Designers parallelize HJOIN?*

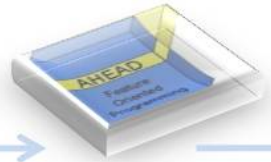
Next Slides Explain Derivation







First Refinement



- Because joins are the most complex operator, increase efficiency by reducing the size of its input streams

- Used Bloom filters to eliminate B tuples that do not join with A tuples

BLOOM Box

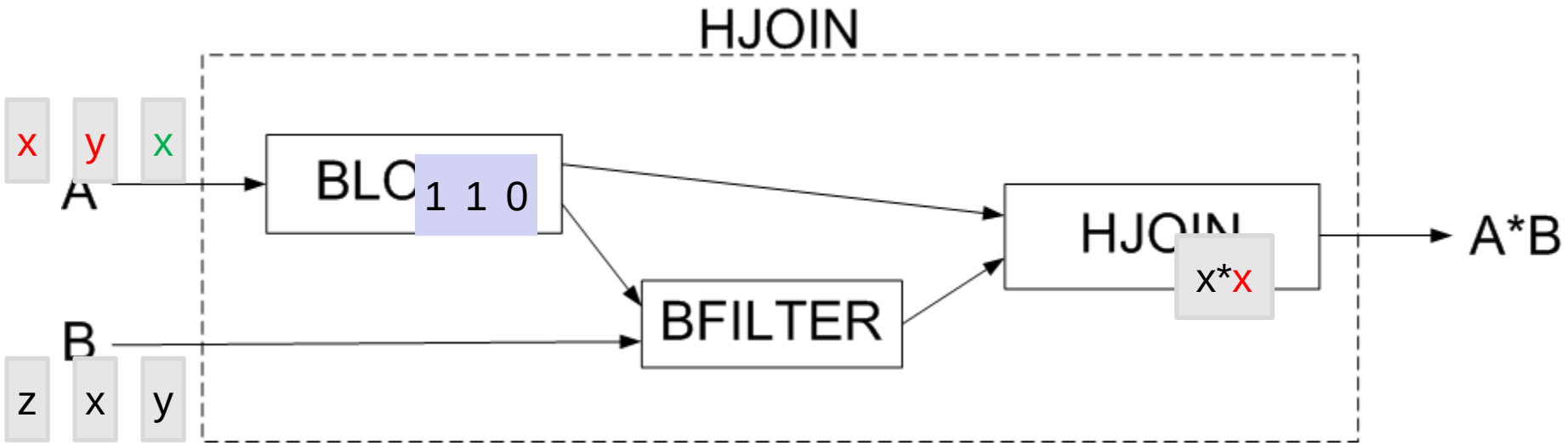
- Bloom filtering is a common technique for disqualifying tuples from further processing

- Algorithm:
  - clear bit map M
  - read each A tuple, hash its key, and mark corresponding bit in M
  - output each tuple A
  - after all A tuples read, output M

BFILTER Box

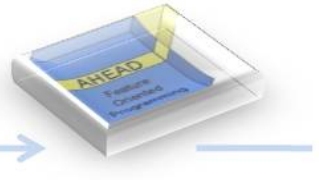
- The filtering part of Bloom filters
  - eliminates B tuples that cannot join with A tuples
- Algorithm:
  - read bit map M
  - read each tuple of B, hash its key: if corresponding bit in M is not set discard tuple (as it will never join with A tuples)
  - else output tuple

The First Refinement

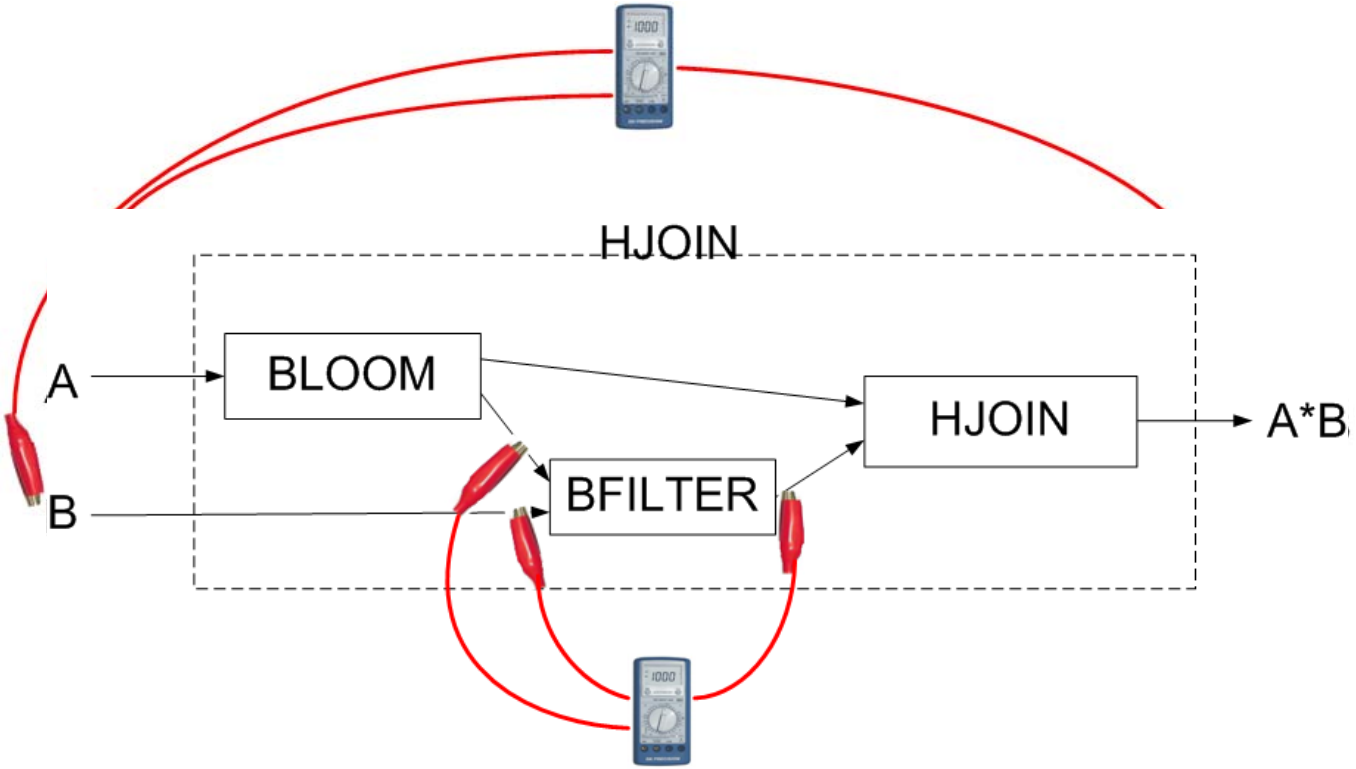


- Expose inner details of HJOIN box
- Can prove correctness of this refinement

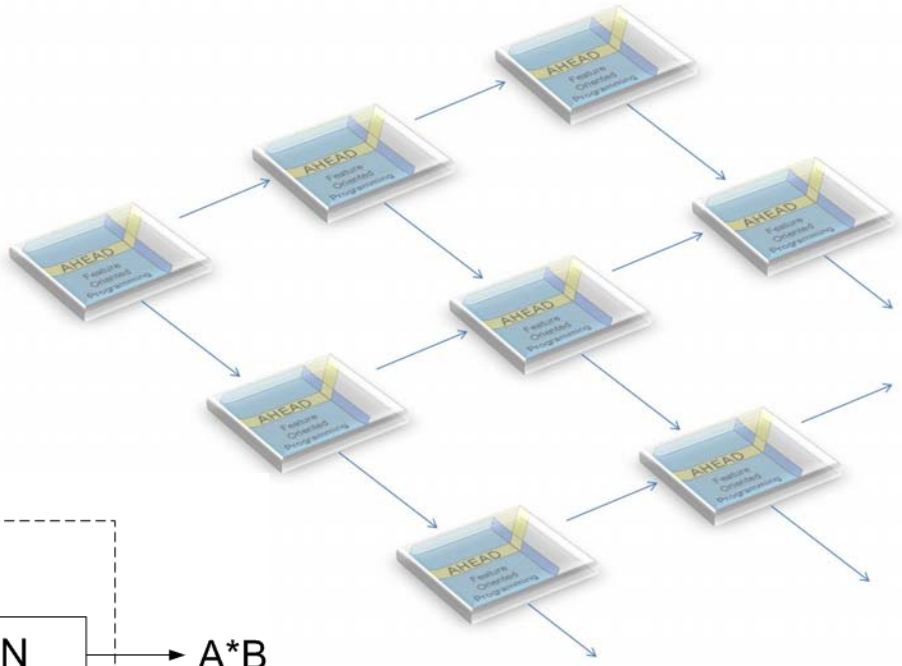
Testing

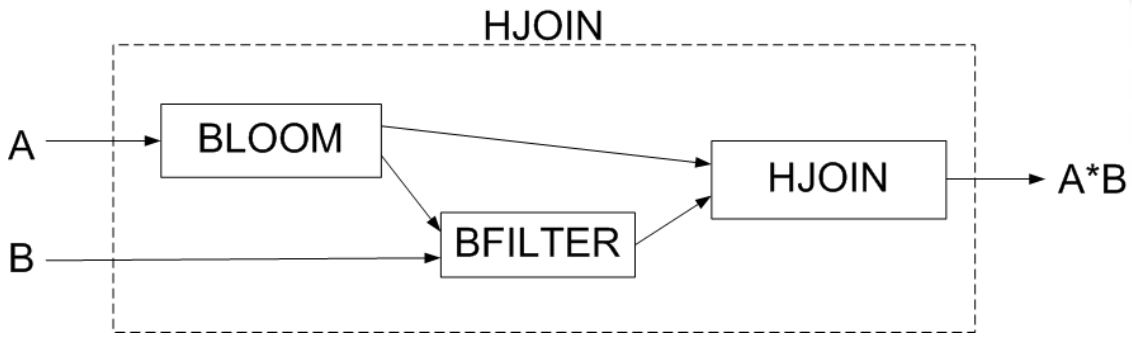


- As we refine, we accumulate and apply tests at every level of abstraction

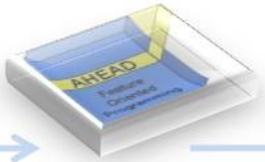


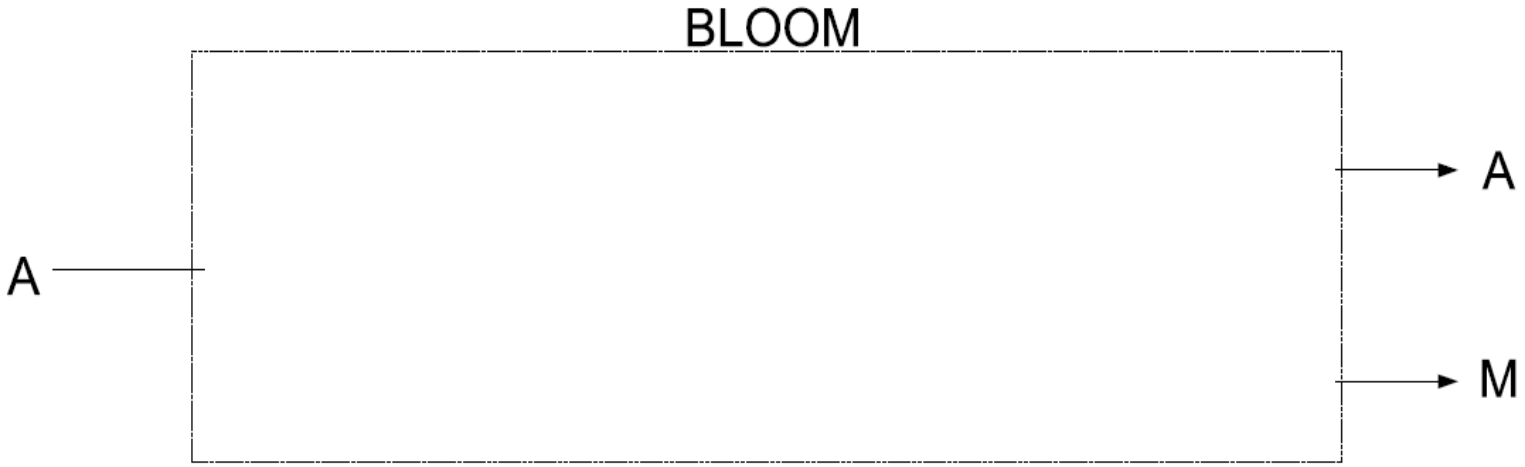
Parallelize Each Box in this Architecture:





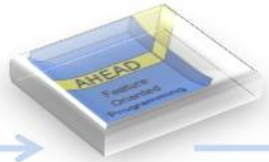
Parallelization of BLOOM Box

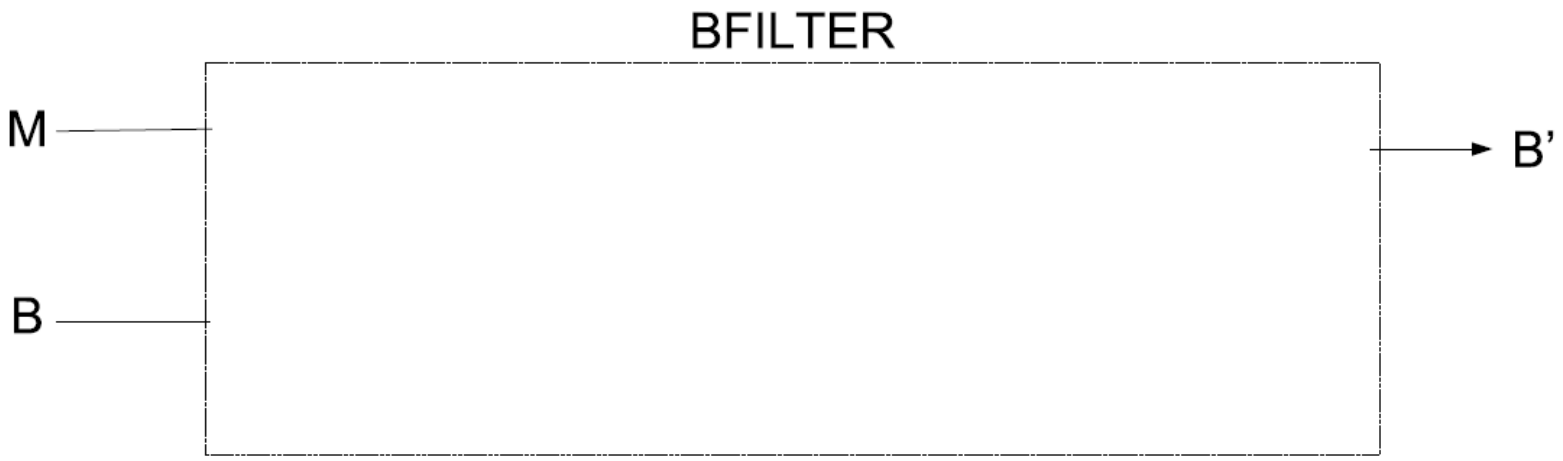




- Algorithm:
  - HSPLIT stream A
  - compute Bloom filter on each substream
  - reconstitute stream A
  - form merge bit maps to produce single bit map M

Parallelization of BFILTER Box

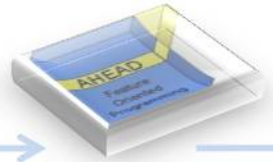




- Algorithm:
  - split M into $M_1 \ldots M_n$ and distribute
  - hash split stream B
  - filter B substreams in parallel
  - reconstitute stream B'

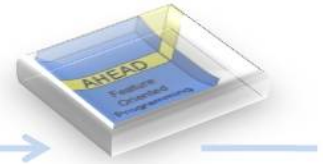
always hash split streams A and B using the same hash function! This gives us properties on which to optimize!

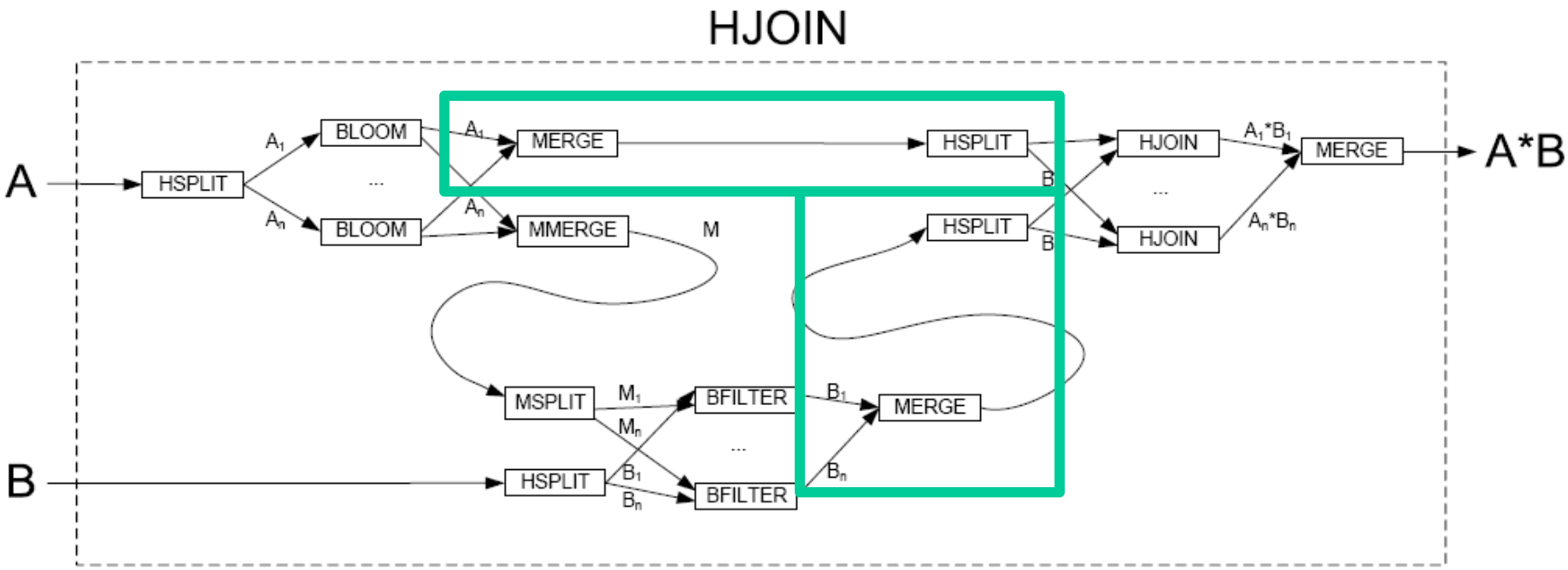
Parallelization of HJOIN Box



- Algorithm:
  - split both streams using same hash function
  - A and B tuples can join only if they have the same hash key
  - perform *n* joins (rather than n^2) in parallel
  - reconstitute join

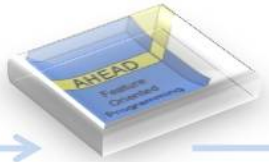
Hierarchical Refinement

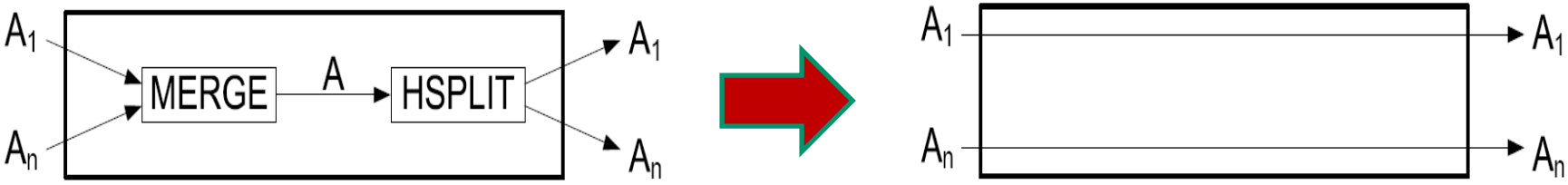




- Substitute parallel implementations for each box
- Note 3 optimizations are possible
- Here are the first two…

A Better View

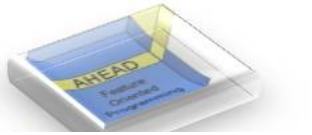


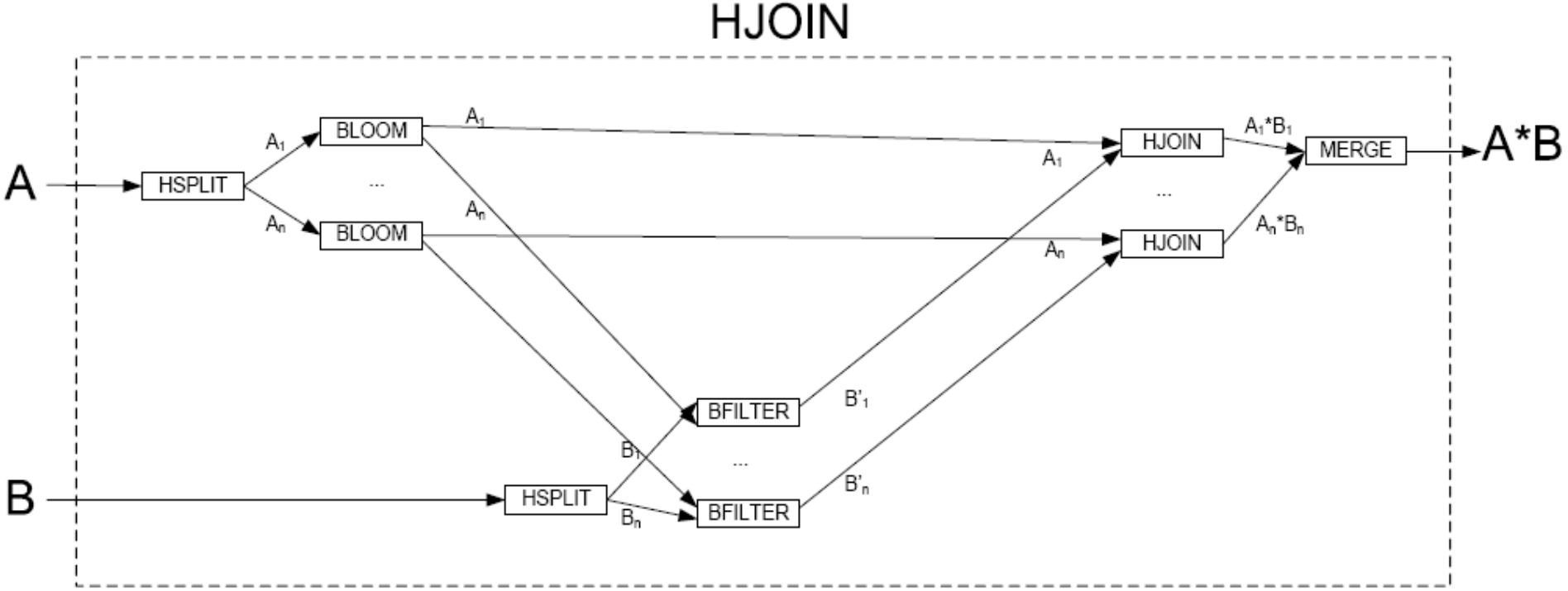


- Stream A is hash split into $A_1 \ldots A_n$, reconstituted, then hash split again
- MERGE – HSPLIT combination is the identity map
- Optimization – get rid of MERGE-HSPLIT

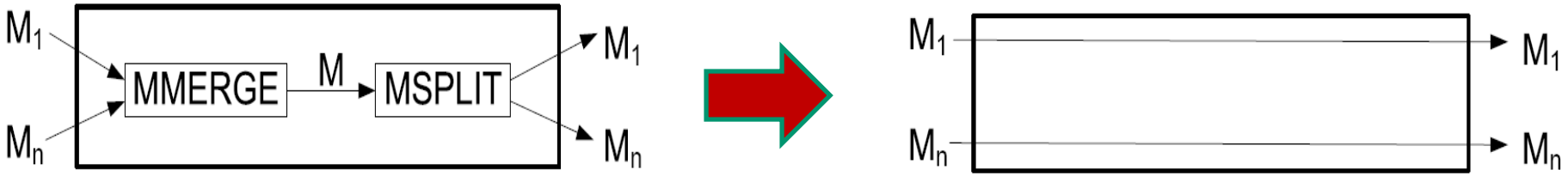
- Same for stream B

Applying Optimizing Rewrite

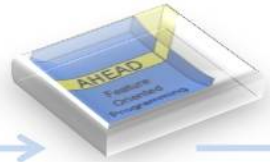


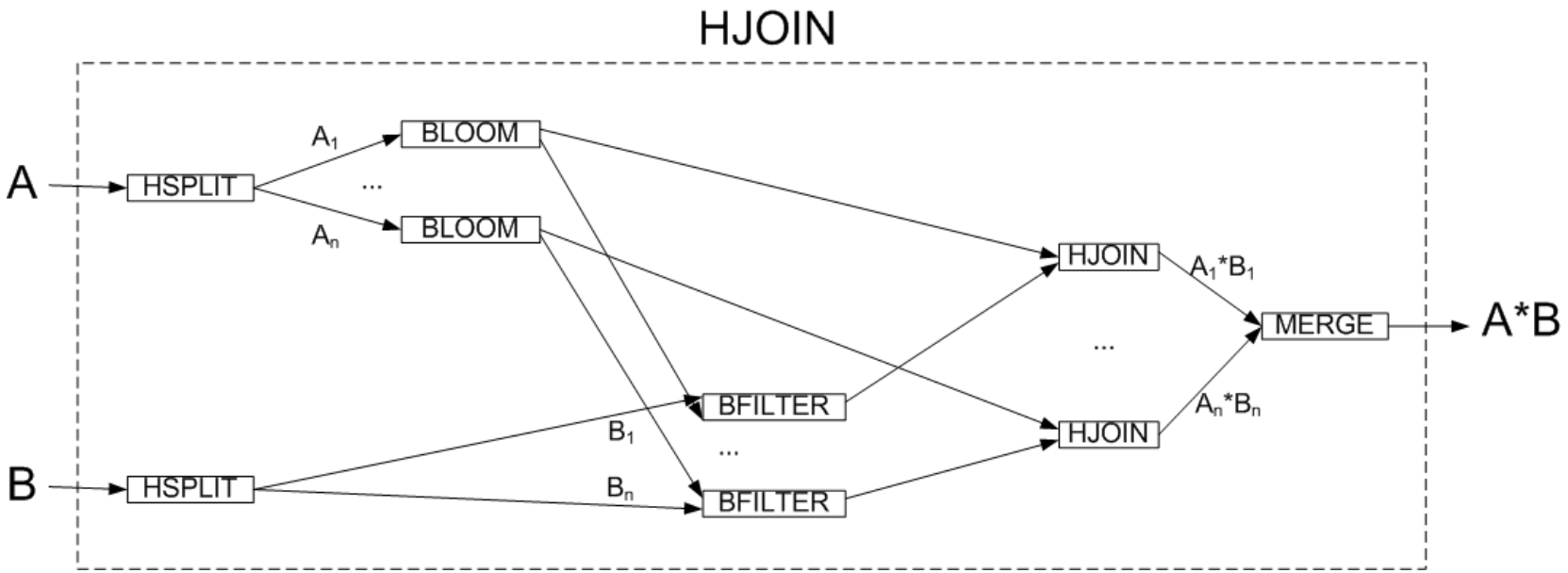


- Still one more optimization to perform…



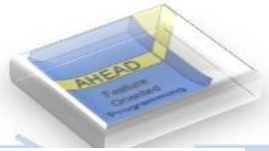
The (Almost) Final Design

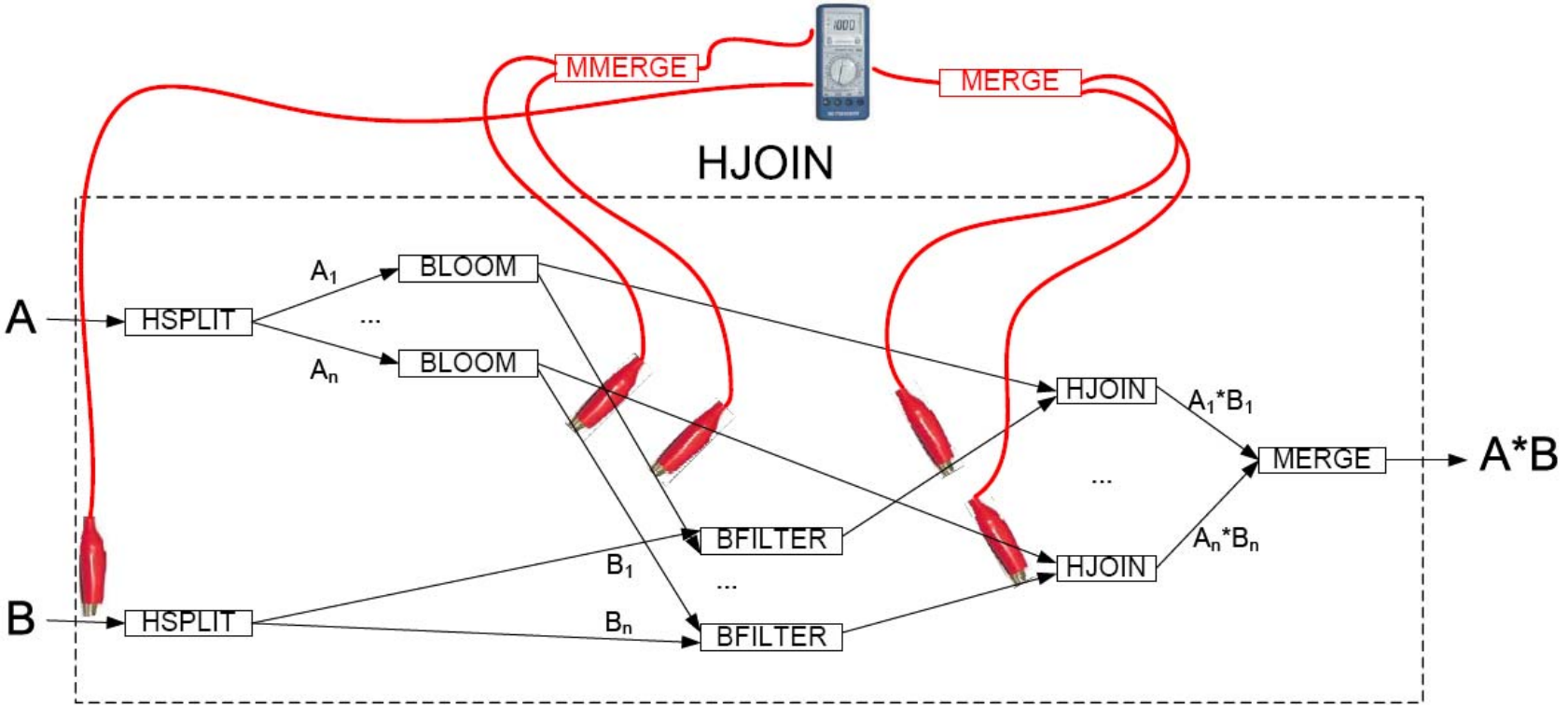




- Elegant
- Easier to remember the derivation than the design itself (!)
- Each step can be proven correct, so the final design is **correct**
- *Not whole picture: exchange rewrites are applied when HJOIN boxes are composed see our paper or original Gamma papers*

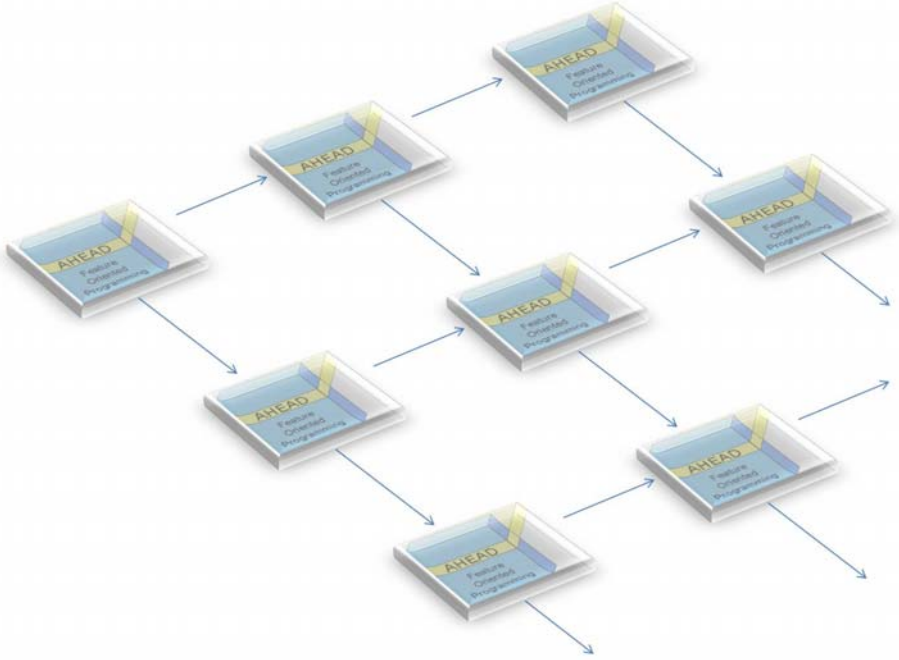
Testing…



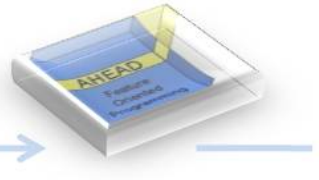


- “Octopus-like” harness for test of BFILTER box
- Again, a lot of engineering is still left, but we have a clear big picture on how to proceed…

#4: Conclusions and Future Work

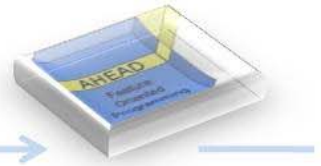


State of Art



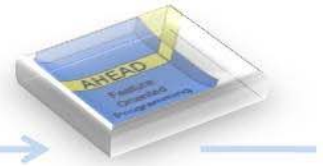
- Experts create and build non-obvious architectures that take time to understand
  - details are accessible to only domain experts and only after a considerable effort to appreciate
  - our experience confirms this, but offers hope…
  - by revealing domain knowledge incrementally in the context of MDE where details of architecture models are progressively revealed, *non-experts can more easily appreciate, participate and contribute in their construction*

Remember

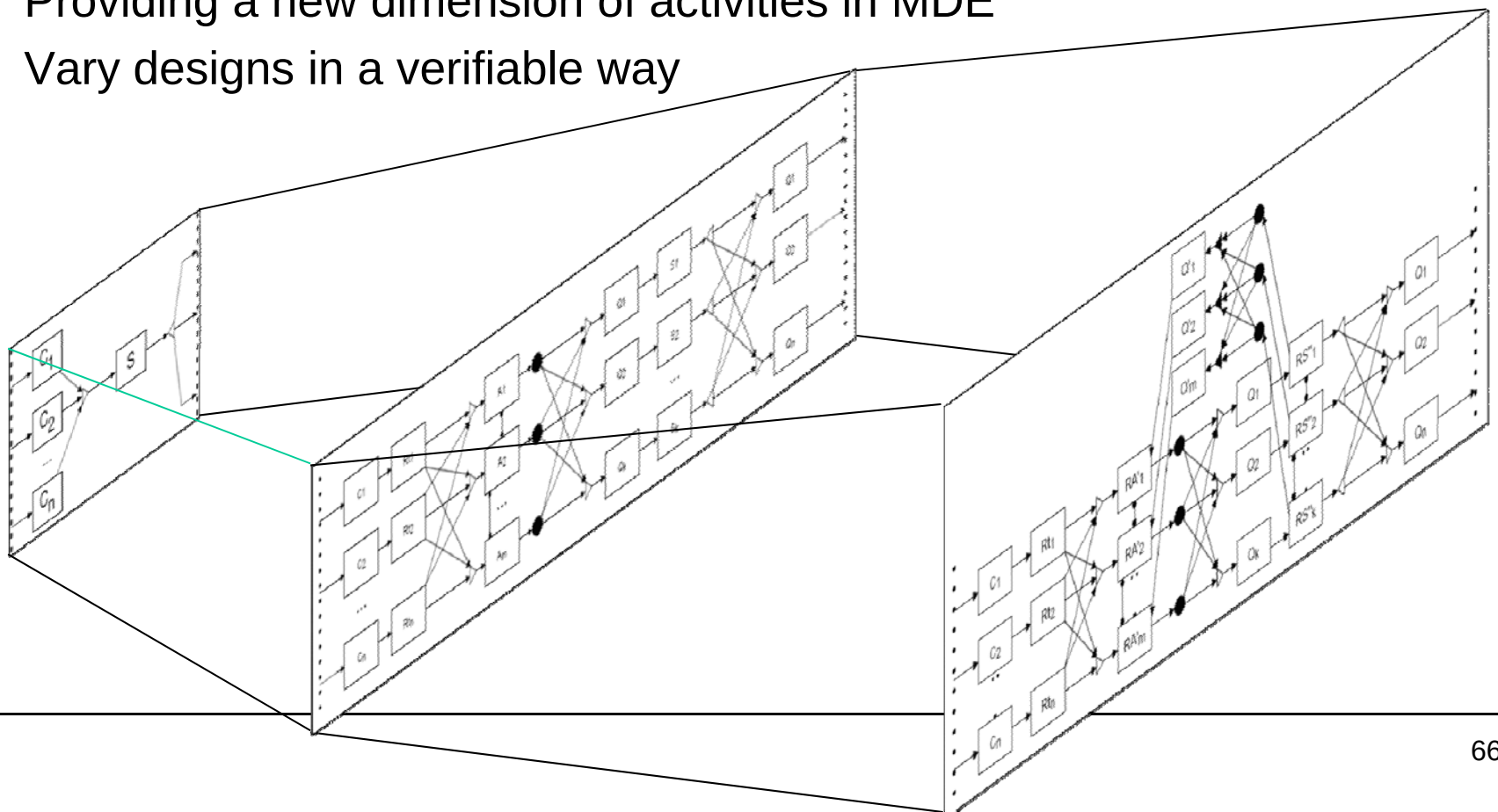


- Don’t create complex designs instantaneously
- Come from simpler designs, recursively
- Showing relationship of simple designs to complex designs enables us to understand and verify our designs, build and test them better
- Ideas are not just “cute” – essential to above goals

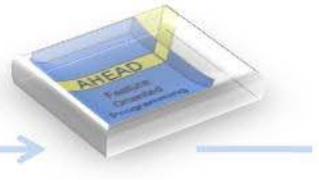
Next Steps



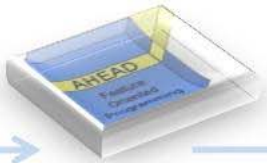
- MDE-based tool set to allow designers to explore interactively a design space by defining and composing transformations
- To be able to synthesize systems from these models
- Providing a new dimension of activities in MDE
- Vary designs in a verifiable way



Closing Observations



- Clear that ideas are being reinvented in different contexts
  - not accidental – evidence we are working toward general paradigm
- Starting down a road that we (community) must travel
  - affords new opportunities to improve “trustworthiness”
  - can verify architectures (which we didn’t have or couldn’t do before)
  - make architecture design, testing, and explanation easier and more structured before
  - expose domain-independent principles underlying software design
- Science of Design



Thank you!

Gracias!