Using Distributed Data Structures for Constructing Cluster-Based Servers

Richard Martin, Kiran Nagaraja
Thu Nguyen and Barbara Ryder

Rutgers University
Department of Computer Science

EASY Workshop
July 2001

Motivation

- Buiding & running large scale internet services is difficult
  - clusters: +isolation, +scalability, –manability, –programming
- Brittle, prone to failure
  - too many components and resulting glue
  - COTS sub-systems not designed for availability
- Large average-to-peak load difference
  - Difficult to shed load gracefully

Approach

- Build services from a set of Data Structures designed specifically for clusters
  - Menu of hashes, lists, and trees
- Compiler-aided analysis for composition of data structures
  - Focus on run-time behavior
- Fault-injection as validation technique
  - Observe reaction under controlled fault conditions

Why Data Structures

- *"It is better to have 100 functions operate on one data structure than have 10 functions operate on 10 data structures"*
  - *Alan J. Perlis*
- Allow service programmer to build services around a few familiar data-structures
- Allow system programmers to deal with hard issues such as replication, fault-tolerance and consistency
- "Collections for a Cluster"

Why Compiler Analysis

- Compiler better at oberserving whole system than programmer
- Encode logic to find dangerous conditions
  - Runtime dangers and static violations
- Analogy:
    - Compiler encodes much performance logic
    - Compiler encodes fault logic as well
    - Reports back to programmer problem areas
- Aid in composition of structures
  - E.g. deciding a good recovery point in program

Why Fault Injection

- Higher confidence in end-to-end system
- Classic testing:
  - Correct input ⇒ correct output
  - Incorrect Input ⇒ report error in input
- Design for faults, use injection to test design
  - Correct input + intermediate error ⇒ recovery or report error

Data Structures: Research Issues

- Can a data structure approach be "easy to use"?
- Difficulty of maintaining uniprocessor abstraction a classic problem
  - trade offs between performance, robustness, uniformity
    - E.g. Hold a remote reference, then remote node dies
- What abstractions balance performability and usability?
- How to compose multiple data structures efficiently?
  - E.g., each structure individually implement a membership protocol?

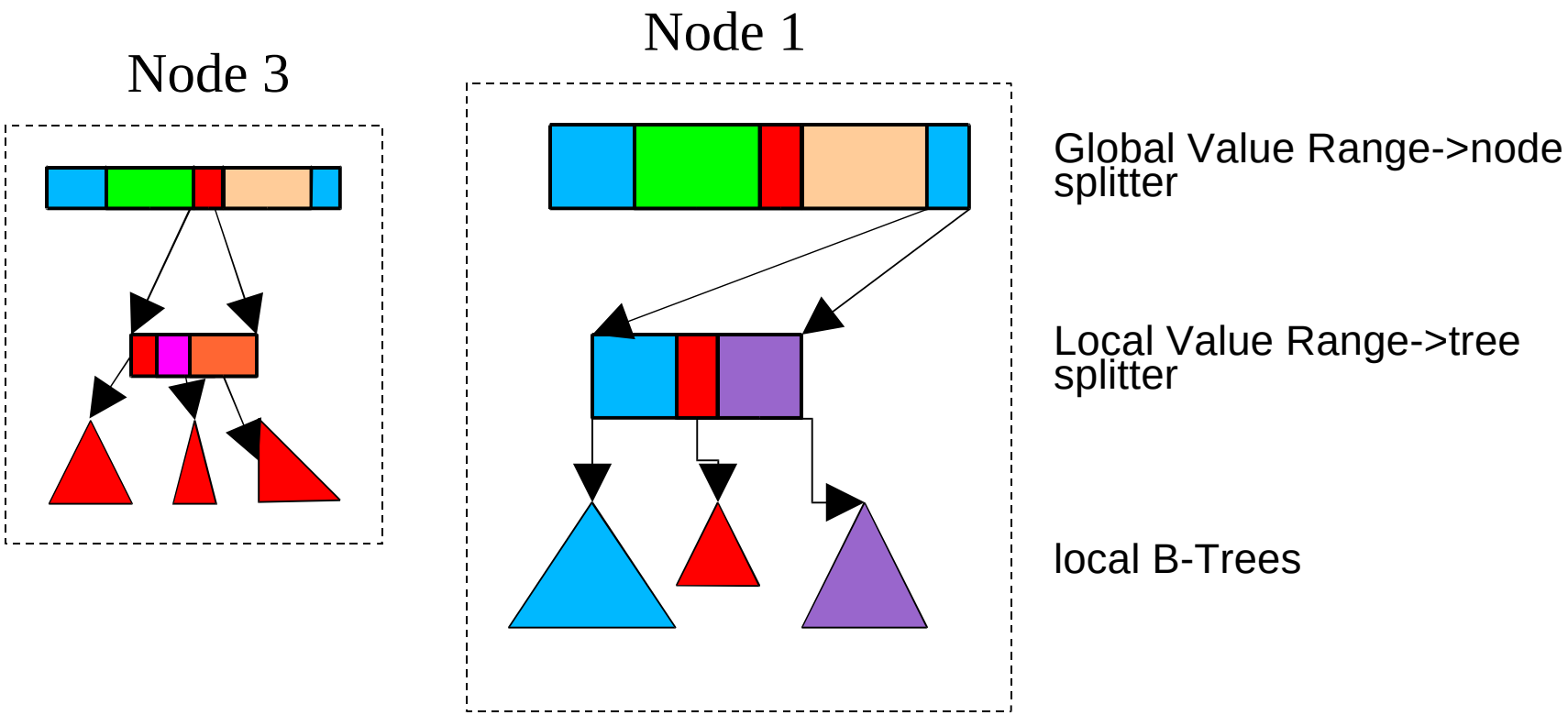
Data Structures: Prototyping Approach

- Use java environment
- Language and run-time system handle tedious programming tasks and balance performance expresiveness
- Java introduces new challenges
  - how to control resources when system hides these details?
  - how to access resources in safe manner through uniform interfaces?

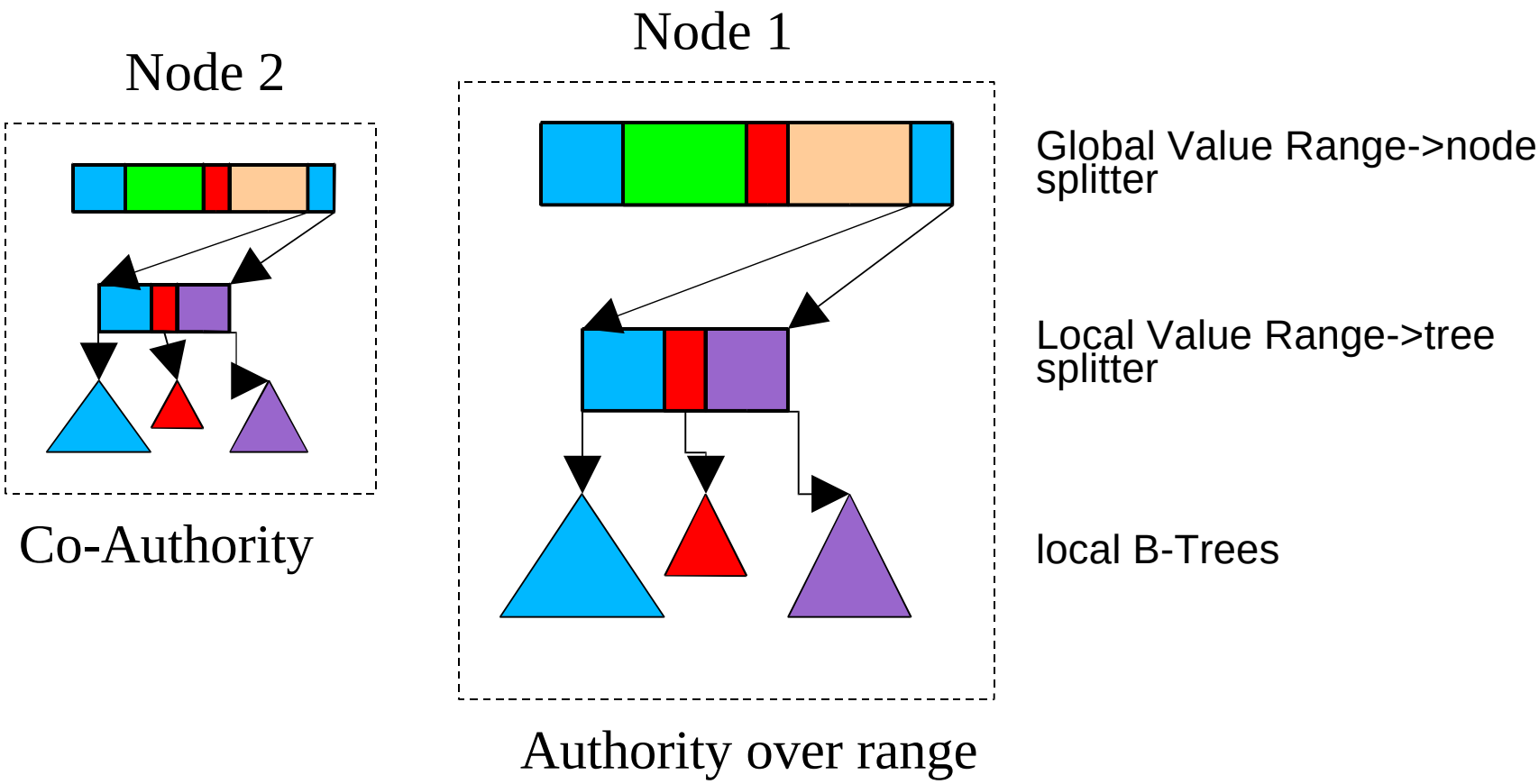
Preliminary Work

- Sorted list
  - Accessible by key & value
  - Iterate over items in sorted order
- Foundation: multiple B-trees per machine
- Meta-data splitter array maintains range info for all nodes
  - fully replicated
  - TRM used to keep consistent

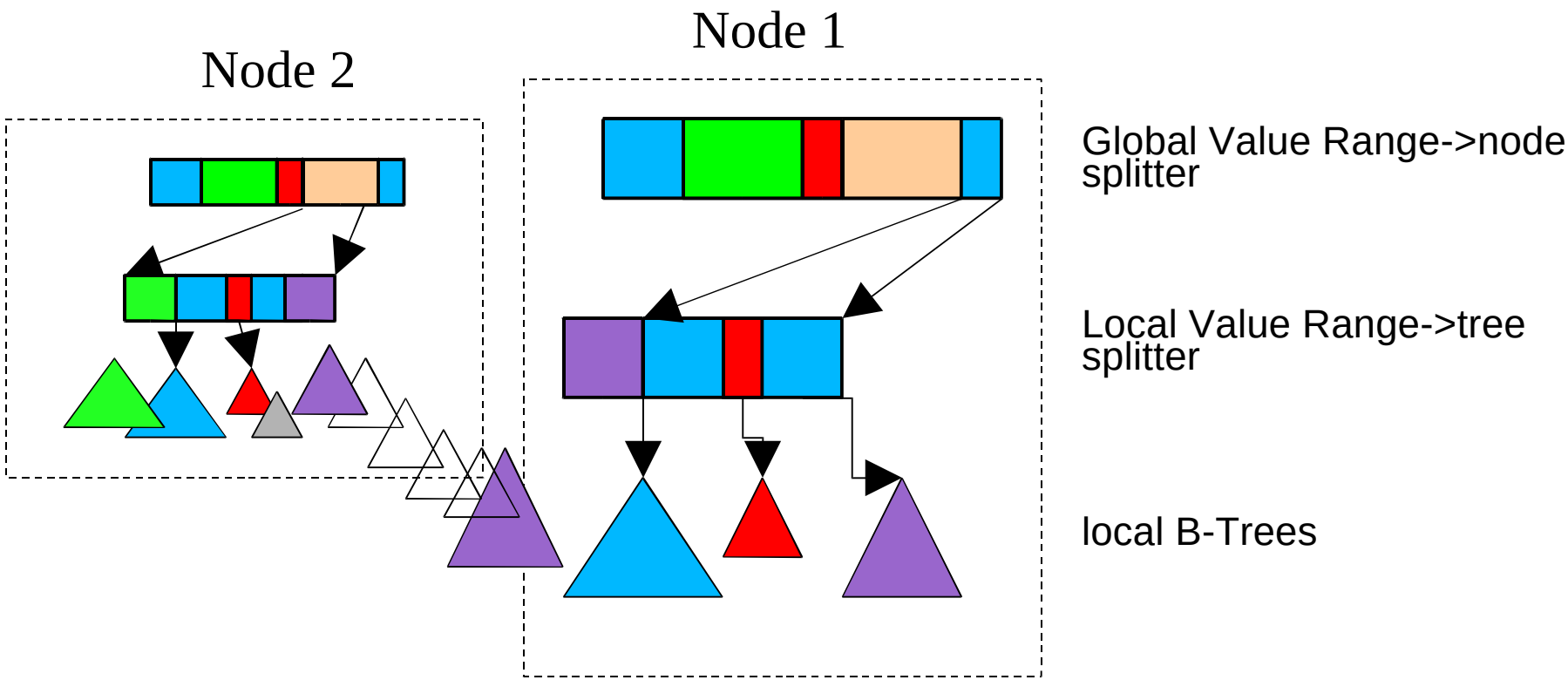
Sorted list: Basic



Sorted list: Replication



Sorted list: Load Balancing



Compiler Analysis

- Types of info: resource exhausion + state violations
  - object escapes (like memory leaks)
  - RMI and JNI calls
  - thread creation/orphans
  - uncaught exceptions
- How to avoid runtime performance penalty?
  - combination of static analysis & dynamic profiling

Fault Injection

- Validate system using fault injection
  - add faults to system, observe response
- What is the upset load?
  - Define componets? I.e. where do components become "too detailed"?
- Where to emulate faults?
  - Exercise differenent components, e.g. a lose a packet in the java runtime? kernel? wires?

Future Directions

- Adaptability
  - allow group to expand/contract
- Recovery
  - What happens when a node recovers?
  - How long to wait before handing off data?
- Composition
  - How to build multiple structures in a single app?