G^2: A Graph Processing System for Diagnosing Distributed Systems

Zhenyu GUO , Dong ZHOU, Haoxiang LIN, Mao YANG,
Fan LONG, Chaoqiang DENG, Changshu LIU, Lidong ZHOU

System Research Group, MSR Asia

Cost of Debugging

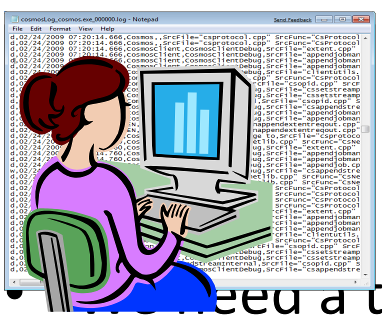
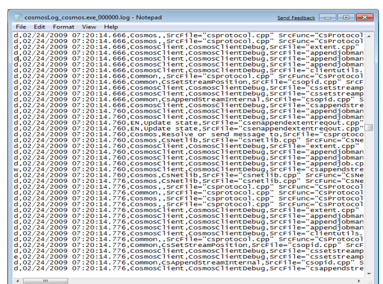
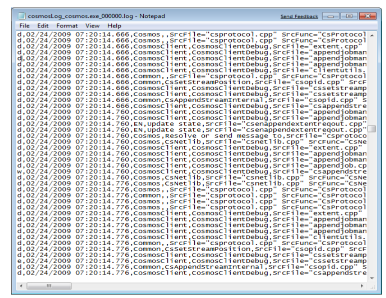
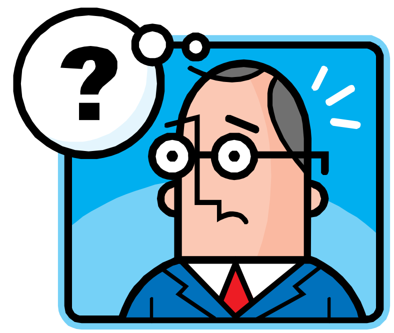
- The huge printing presses for a major Chicago newspaper began malfunctioning …

$10,000 = $1 + $9,999

Most bugs can be fixed quickly,
however identifying the root causes is hard.

Motivation

- Diagnosing distributed systems is frustrating
  - Execution is too complex to comprehend
  - Tons of logs, but correlations are missing
  - Lost in the information sea

- We need a tool that
  - Finds correlated information.
  - Facilitates better summarization and reasoning
  - Is fast and easy to use

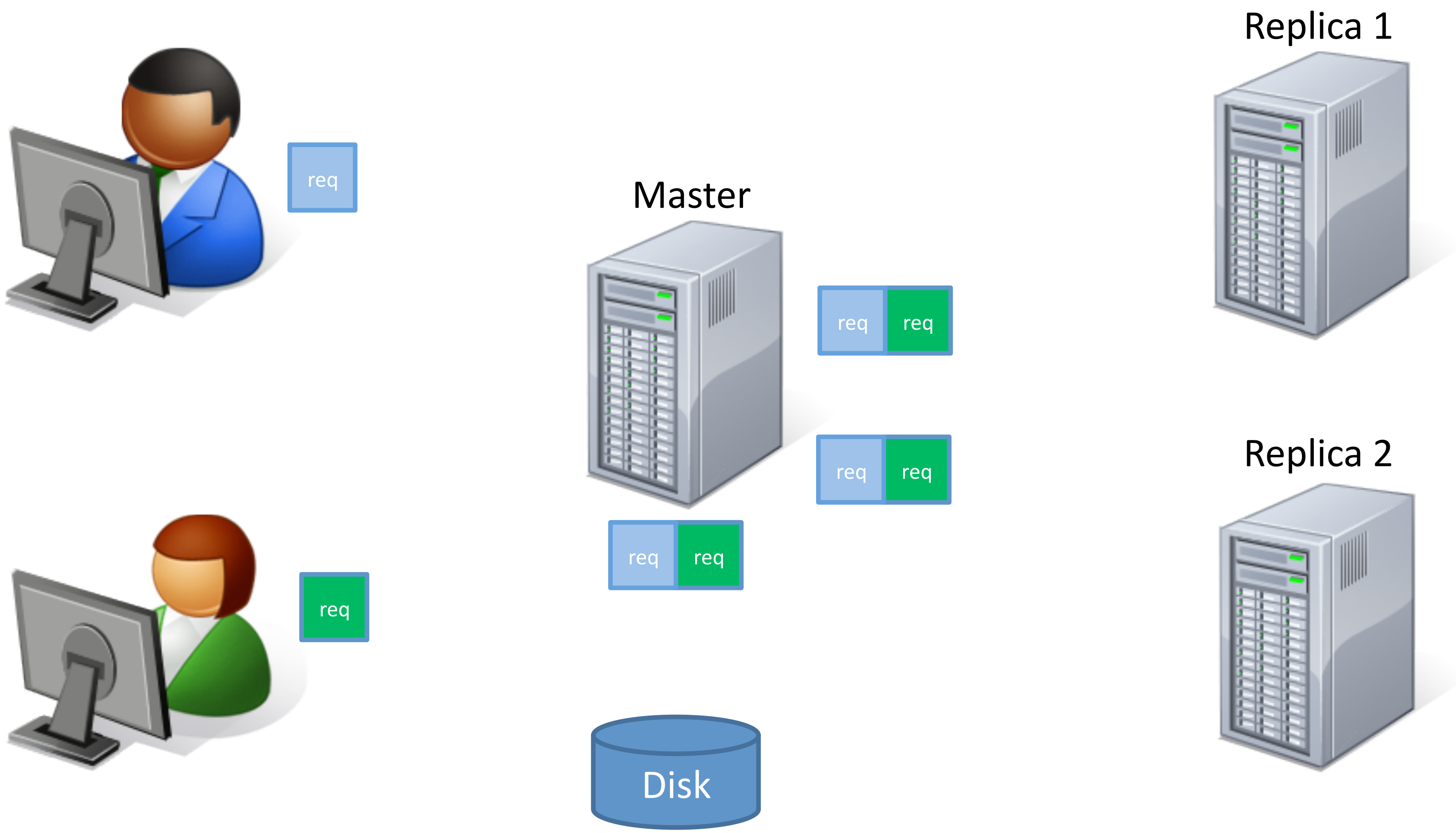
Contribution

- Graph based diagnosis for distributed systems
  - Execution graph to capture correlations
  - Graph based diagnosis operators
    - **Slicing** for finding & filtering
    - **HierarchicalAggregation** for summarization
- Declarative diagnosis queries
  - Integrated with Microsoft LINQ
- Distributed engine
  - Integrated relational computation and graph traversal
  - Optimizations based on the characteristics of the execution graph and diagnosis operators

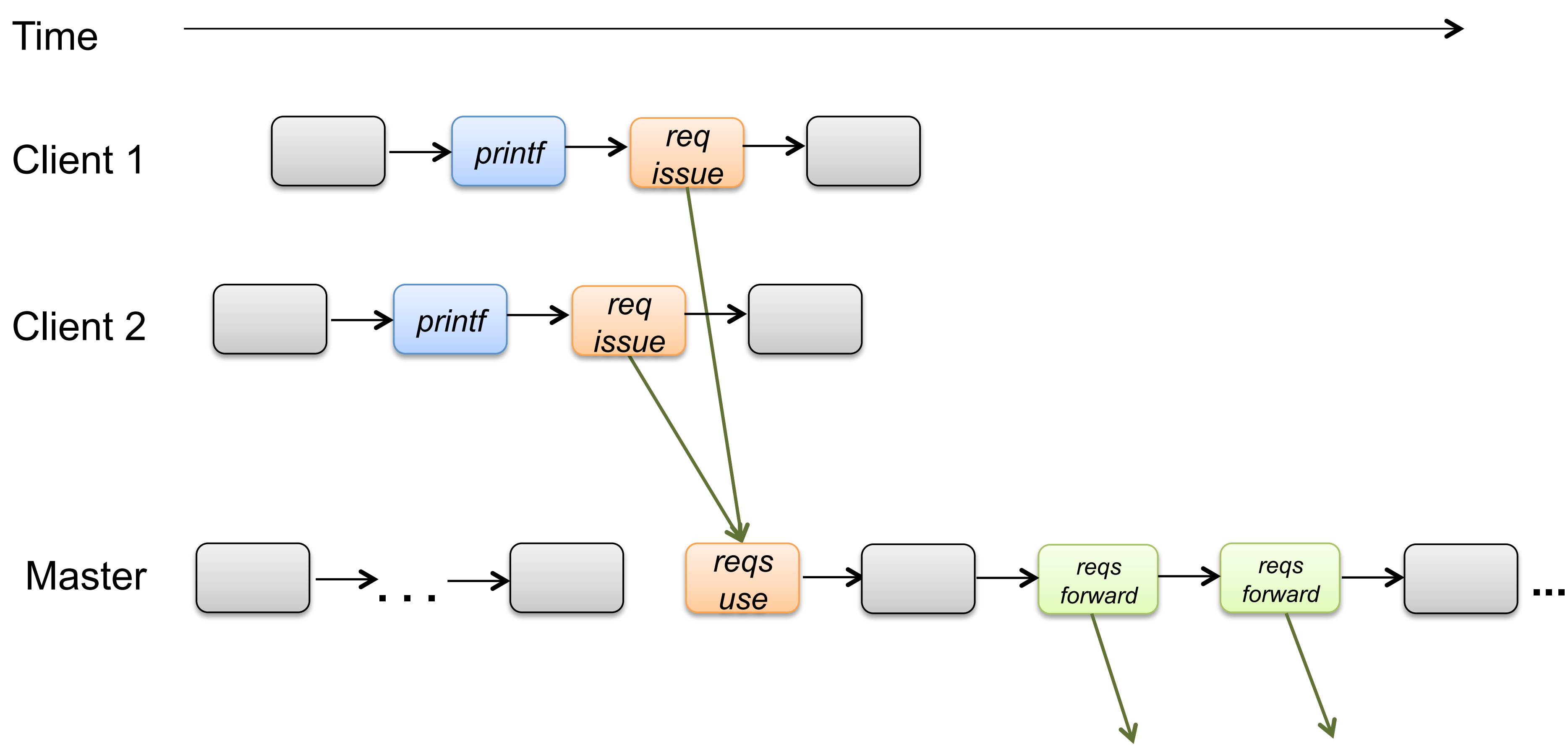
Outline

- Model
- Engine
- Programming
- Evaluation

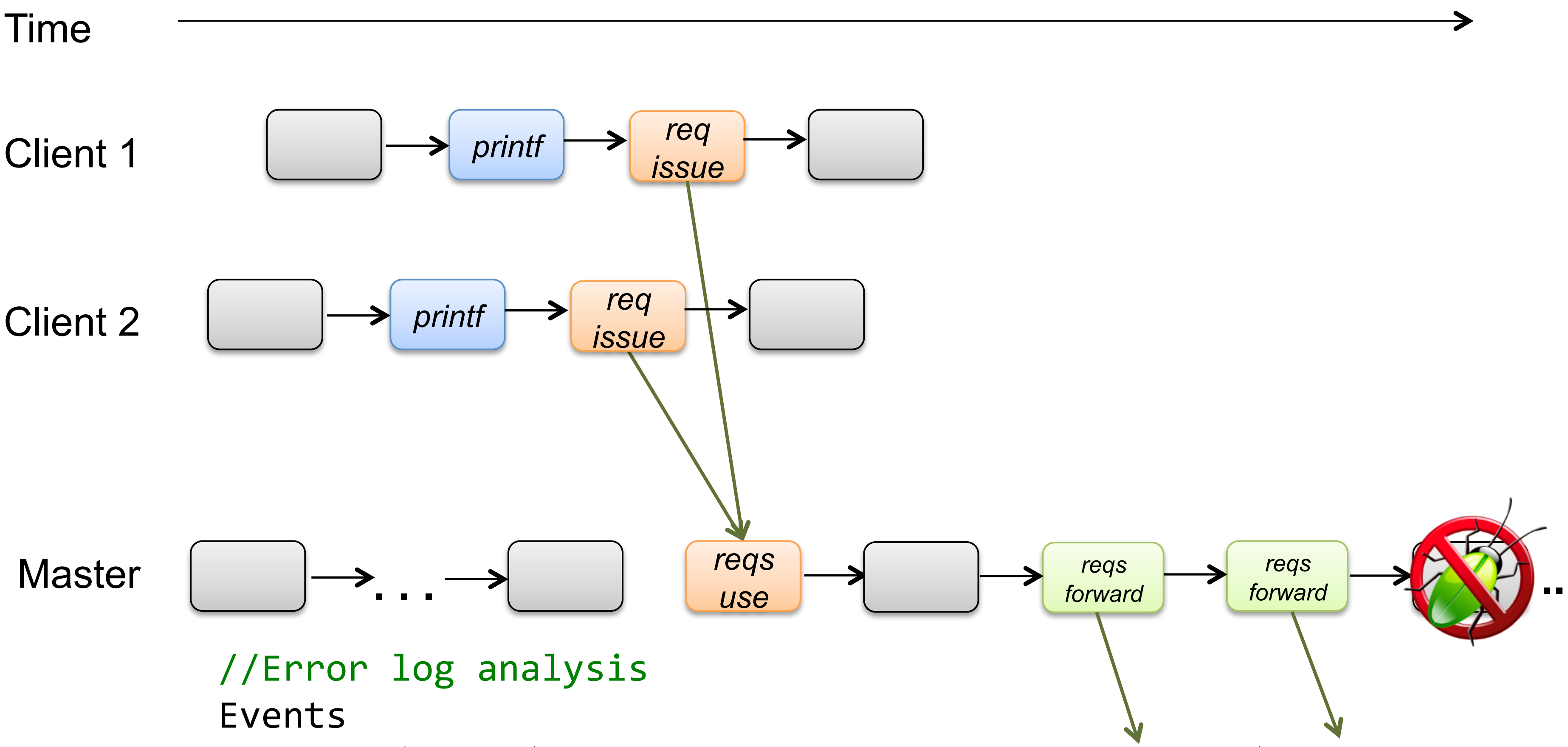
Capture Correlations



Execution is Graph

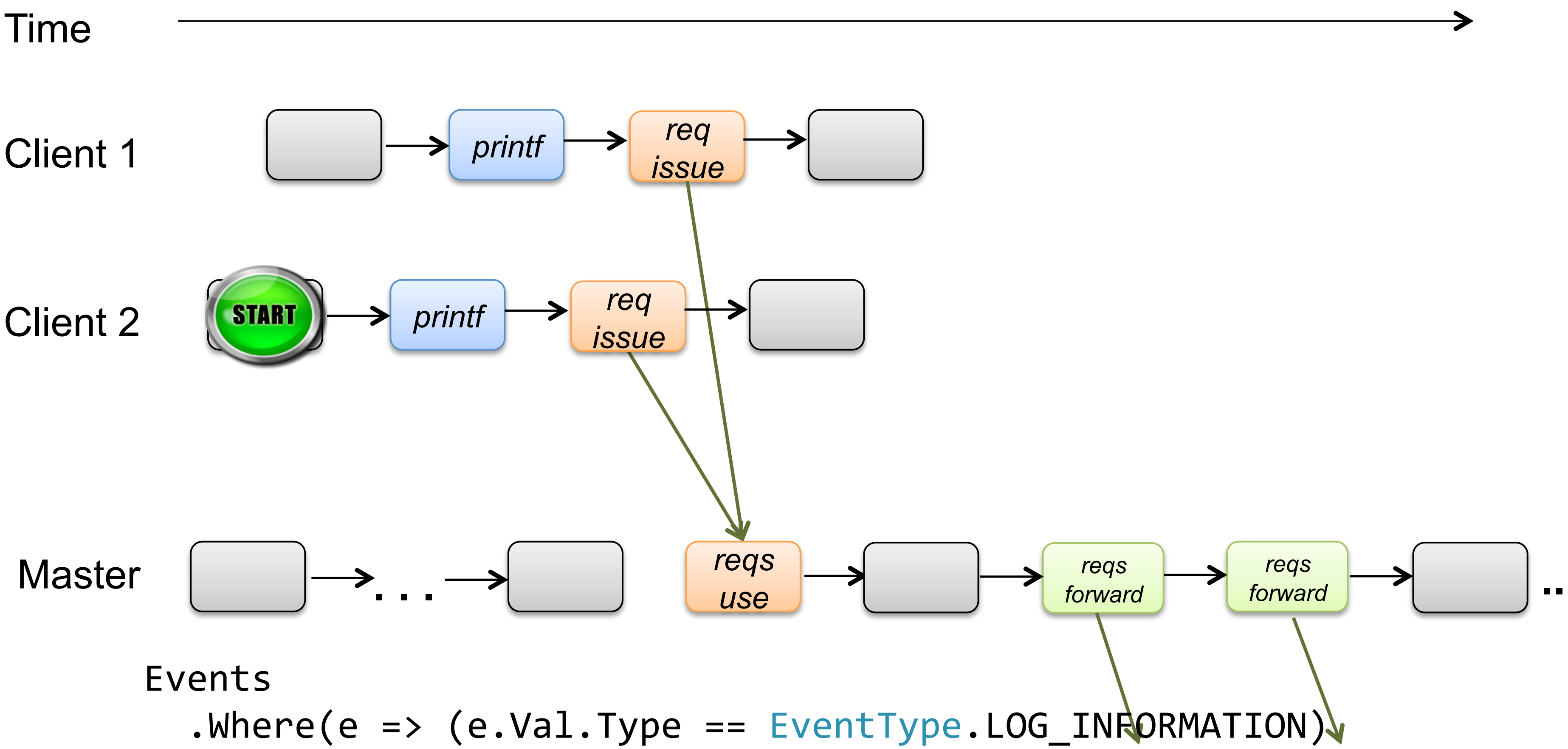


Slicing: Find the correlated subgraph and filter others by traversing the execution graph



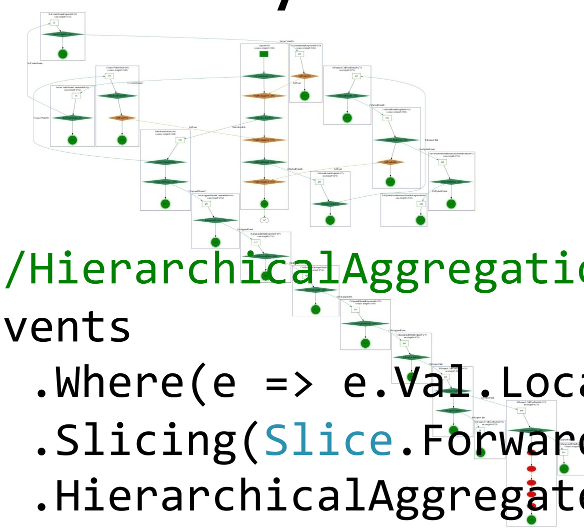

```
//Error log analysis
Events
   .Where(e => (e.Val.Type == EventType.LOG_ERROR)
         && e.Val.Payload.Contains("Write request failed"))
   .Slicing(Slice.Backward)
   .Select(e => Console.WriteLine(e.Val.Payload));
```

Slicing: Find the correlated subgraph and filter others by traversing the execution graph



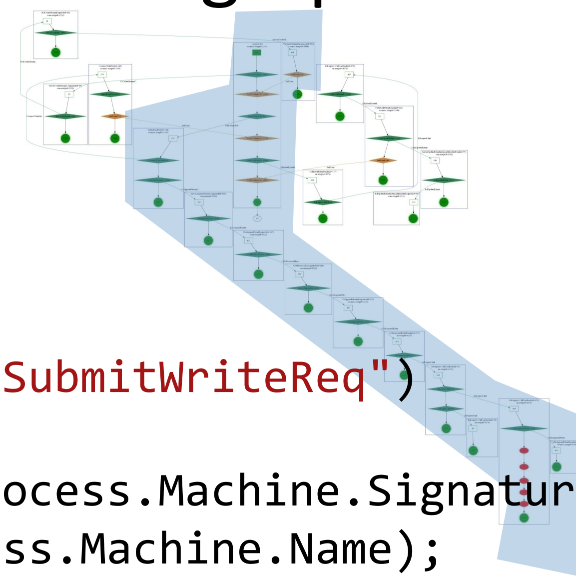

```
Events
  .Where(e => (e.Val.Type == EventType.LOG_INFORMATION)
        && e.Val.Payload.Contains(“Start ClientRequest()"))
  .Slicing(Slice.Forward)
  .Select(e => Console.WriteLine(e.Val.Payload));
```

HierarchicalAggregation: Summarize details by traversing the execution graph

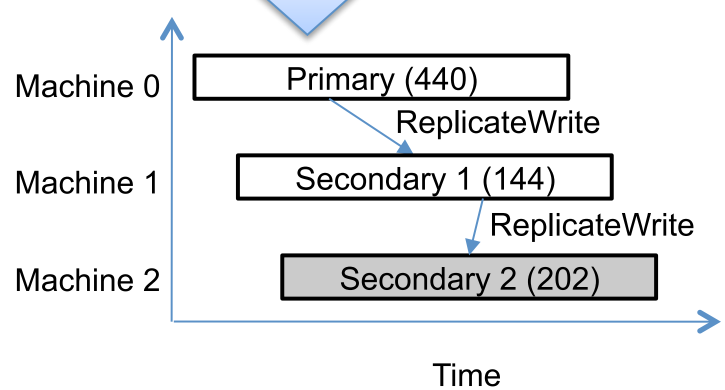


```
//HierarchicalAggregation
Events
  .Where(e => e.Val.Location.Name == "SubmitWriteReq")
  .Slicing(Slice.Forward)
  .HierarchicalAggregate(e => e.Val.Process.Machine.Signature,
        evts => evts.First().Val.Process.Machine.Name);
```

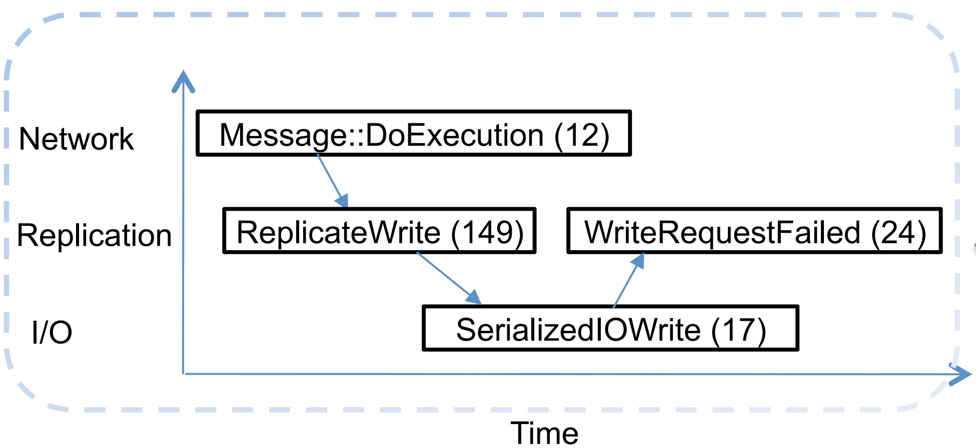
Slicing



Aggregation



Zoom In



Understand Execution Graph

- Execution graph is rather huge
  - A 2-hour SCOPE/Dryad graph has over 1.2 billion vertices, 0.54 billion edges, and lots of user payload(logs)
- Connected subgraph is also huge
  - However, intra-machine interactions are much more than inter-machine ones(91% vs 9% in SCOPE/Dryad graph)
- Graph structure data is relatively small
  - User payload is over 64% in storage
- Iterative access to graph structure data
  - Concurrent traversals
  - Aggregation follows slicing

Optimize Graph Access

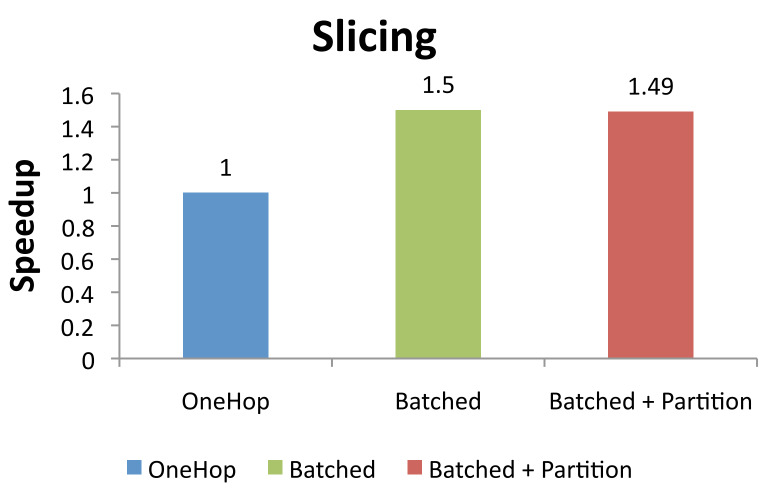
- Diagnosing tool as a distributed system
- Optimal partition on graph data
  - At machine boundary initially. Dynamic partitioning.
  - Local data is stored in database
- Caching
  - Graph structure data in memory
  - Retrieve payload only when necessary
- Prefetching
  - Get vertex properties during slicing, instead of during aggregation

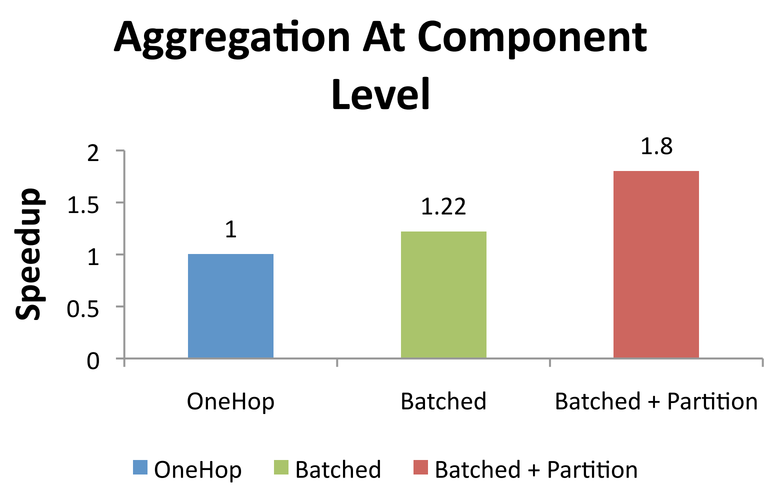
Understand Slicing & HierarchicalAggregation

- Latency is an issue
  - More than 200 hops sometimes, due to deep paths
- Rigorous synchronization is not efficient
  - Different from Page Rank/Belief Propagation
- Aggregation repeatedly colors local vertices with the same aggregation identity
  - Lots of local messages

Optimize Fast Execution Graph Traversal

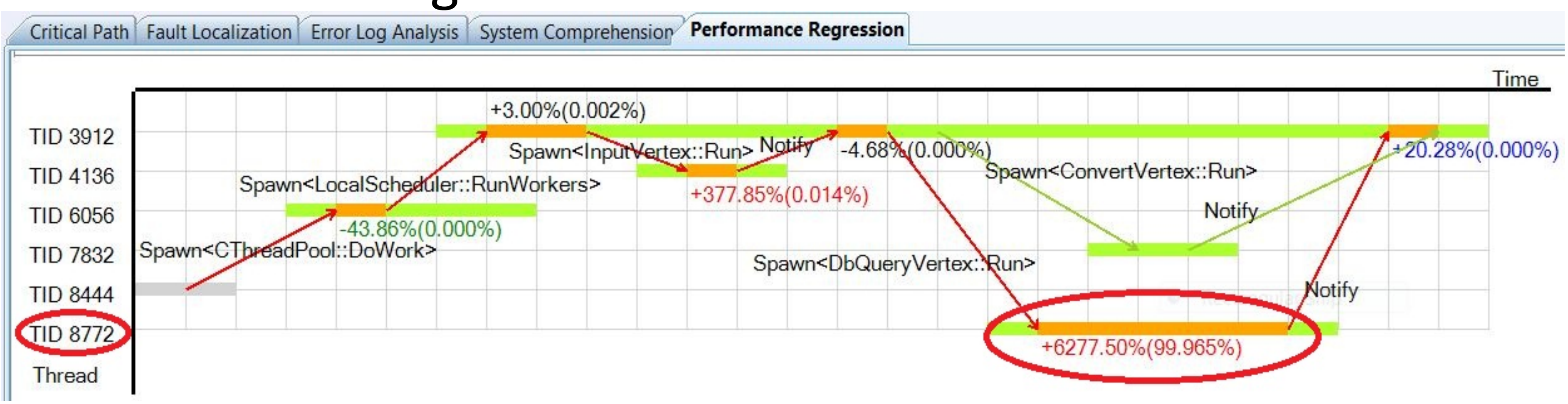
- Batched Asynchronous Graph Traversal
  - Explore local vertices until reaching cross-partition edges without synchronization
- Partition-level interface
  - One traversal worker on each partition
  - Direct access to the whole local graph data
  - Local vertices could be condensed into super nodes in advance





Play with G^2

- Capture the graph
  - Manual annotation, Binary rewriter and dynamic instrumentation
- Write simple C# queries
  - Reuse existing relational operators in LINQ
  - Slicing(Chopping) / HierarchicalAggregation
  - Local Extensions: Diff, CriticalPath, …
- Provide diagnosis wizards in Visual Studio



Evaluation

Systems	LOC(K)	Func#	Edge#	Event#	Raw(MB)	DB(MB)	Time(min)	Node#
Berkeley DB	172	46164	92502	186597	14	29	2	3
G^2	27	267,728	634,704	1,212,778	85	231	17	60
SCOPE/Dryad	1,577	3,128,105	8,964,168	20,106,457	1,226	3,269	120	60

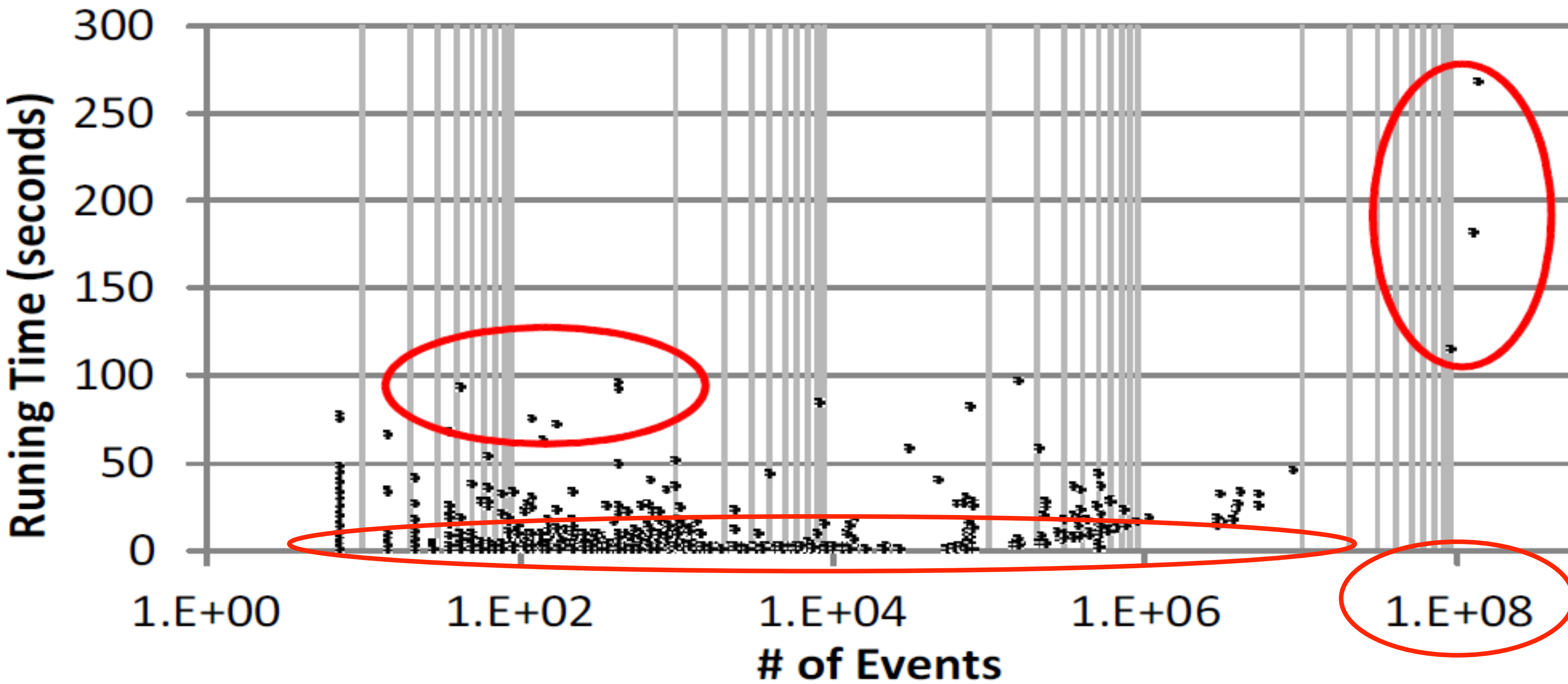
Table 1: Per node graph statistics

Systems	Annotated Edge#	Annotated CS#	Instrumented Func#	Rules
Berkeley DB	2	2	1,542	23
G^2	9	11	197	10
SCOPE/Dryad	17	13	730	5

Table 2: Instrumentation statistics

- 60 machines
- 2 GHZ dual core
- 8 GB memory
- Two 1 TB disk
- 1 Gb Ethernet

End to End Query Performance



5770 random queries on the SCOPE/Dryad Graph

```
Events.Where (e => …)
        .Slicing(Slice.Forward)
        .HierarchicalAggregate(e => e.Val.Process.ID);
```

Related Work

- Execution Model
  - Path based analysis
  - Pure log analysis
  - Static analysis
- Distributed Execution Engine and Storage
  - Graph systems
  - Map-reduce alike systems
- Diagnosis Platform
  - Cloud9: Testing as a service
  - Dapper: path analysis atop of BigTable

Conclusion

- Graph based diagnosis for distributed systems
  - Slicing for filtering the logs
  - HierarchicalAggregation for summarization
- Graph engine with specific requirements
  - Integrated relational computation and graph traversal support
  - Batched asynchronous graph traversal and partition-level interface for better performance

Thanks!

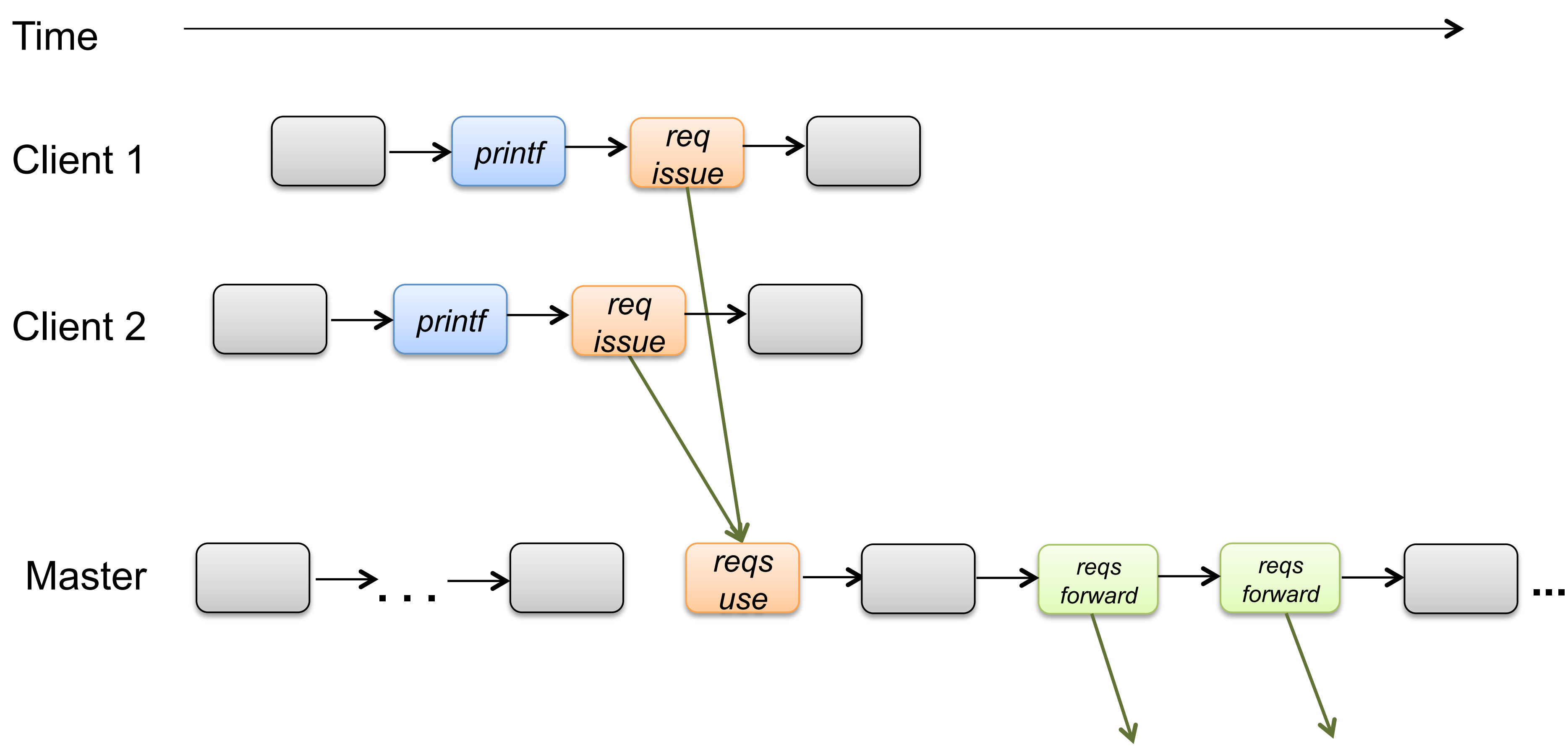
Generations for log structure and related tools:

Text

- Unstructured text
  - Format: [***AUTO****: time, component, log level, pid, tid, location*], `printf' message
  - Aggregation: by the meta information or keywords in the unstructured message
- Pros
  - Free style format
  - Easy to process: grep
- Cons
  - May miss many implicit dependencies among log entries without shared tag (e.g., request id)

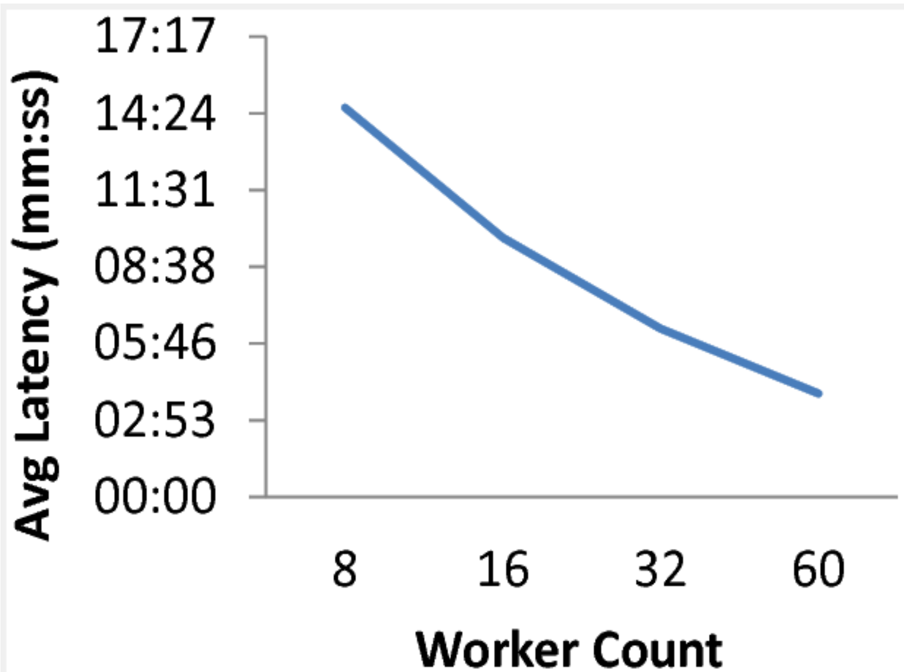
Generations for log structure and related tools:
Paths

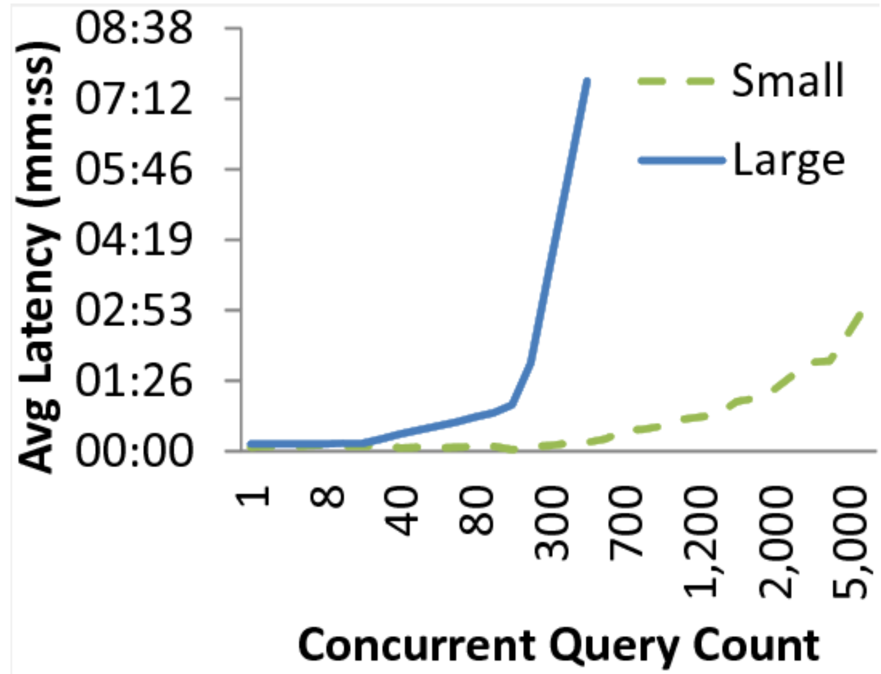
- Path-based aggregation
  - Format:
    - **[*ANNOTATION:*** *path id***]** + unstructured text
    - Optional: **[*ANNOTATION*]** dependencies among log entries belonging to the same path are captured
  - Aggregation: by the user request id (path id)
- Pros
  - Effective for request-centric analysis and modeling
  - The logs are partitioned by request id, and each partition can usually be handled by single machine
  - A nice balance between usability and the overhead
- Cons
  - Cut off interactions between requests, which is common in distributed systems, such as batching
  - Path is statically defined by the pre-defined `requests' only



Time
Client 1
printf
req issue
Client 2
printf
req issue
Master
. . .
reqs use
reqs forward
reqs forward
...

Scaling Performance



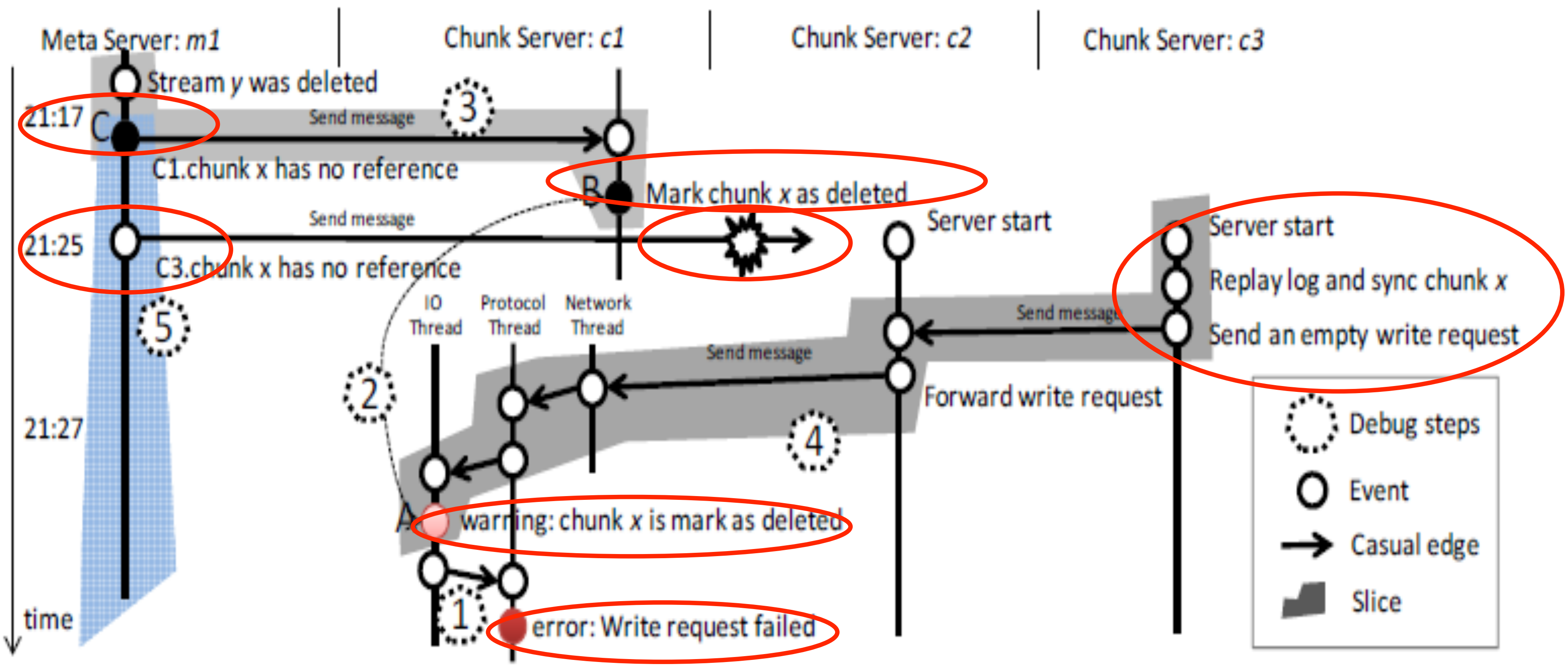


Graph Traversal Interface

```
IQueryable<T> GraphTraversal<TWorker>(
        this Graph<TV, TE> g,
        IQueryable<Vertex<TV, TE>> startVertices
        ) where TWorker : GPartitionWorker<TV, TE, _, T>;


class GPartitionWorker<TV, TE, TMsg, T>
{
    public Vertex<TV, TE> GetLocalVertex(ID VertexID);
    public void SendMessage(ID VertexID, TMsg msg);
    public void WriteOutput(T val);
    public virtual void Initialize(VertexIterator<TV, TE>);
    public virtual void OnMessage(Vertex<TV, TE>, TMsg msg);
    public virtual void Finalize();
}
```

```
class GPartitionSlicingWorker<TV, TE> : GPartitionWorker<TV, TE, bool, Vertex<TV, TE>>
{
    private HashSet<ID> VisitedVertices;

    public override void  Initialize(VertexIterator<TV,TE> inits)
    {
         foreach (var v in inits)
         {
             SendMessage(v.ID, true);
         }
    }

    public override void  OnMessage(Vertex<TV,TE> v, bool msg)
    {
        if (VisitedVertices.Contains(v.ID)) return;
        VisitedVertices.Add(v.ID);
        WriteOutput(v);
        foreach (var e in v.OutEdges)
        {
            if (e.IsCausal())
                SendMessage(e.DstVertexID, true);
        }
    }
}
```

Experience using G^2



Deployment Issues

- Capture the correlations
  - Instrument the network and thread pool libraries to capture the asynchronous transitions among threads and machines
- Store and process the logs
  - Option 1: dedicated graph engine (G^2)
    - Pros: complete support of G^2 diagnosis queries
    - Cons: interference to host systems
  - Option 2: in-app graph engine with latest logs
    - Pros: lightweight, easy to deploy
    - Cons: limited memory cache capacity (latest logs only)