AGENDA

- The Need for IMDG-Based Data Structures:
  - Brief Review of Using In-Memory Data Grids (IMDGs) for Data Access & Query
  - Computing in the Client: the Network Bottleneck
- Build Data Structures in the Grid:
  - Using the Grid as an Extensible Data-Structure Store (e.g., Redis)
  - Challenges with Running in the Grid Service
- Build Extensible Data Structures Outside the Grid Service:
  - Example of “Single Method Invocation” (SMI)
- The Next Step: Data-Parallel Data Structures and Methods:
  - Example of “Parallel Method Invocation” (PMI)
  - Sharding Data Structures for Data-Parallelism
- Combining Single and Data-Parallel Operations
ABOUT THE SPEAKER

- Dr. William Bain, Founder & CEO of ScaleOut Software:
  - Ph.D. in Electrical Engineering (Rice University, 1978)
  - Career focused on parallel computing – Bell Labs, Intel, Microsoft
  - 3 prior start-ups, last acquired by Microsoft and product now ships as Network Load Balancing in Windows Server

- ScaleOut Software develops and markets In-Memory Data Grids, software middleware for:
  - Scaling application performance and
  - Providing operational intelligence using
  - In-memory data storage and computing

- Eleven years in the market; 425+ customers, 10,000+ servers
THE NEED FOR IMDG-BASED DATA STRUCTURES
WHAT ARE IN-MEMORY DATA GRIDS?

- In-memory data grid (IMDG) provides **scalable, high storage for live data**:
  - Designed to manage business logic state:
    - Sequentially consistent data shared by multiple clients
    - Object-oriented collections by type
    - Create/read/update/delete APIs for Java/C#/C++
    - Parallel query by object properties
  - Designed for **transparent** scalability and high availability:
    - Automatic load-balancing across commodity servers
    - Automatic data replication, failure detection, and recovery
  - IMDG also provides an ideal platform for “operational intelligence”:
    - Easy to track live systems with large workloads
    - Appropriate availability model for production deployments
COMPARING IMDGS TO SPARK

- On the surface, both are surprisingly similar:
  - Both designed as scalable, in-memory computing platforms
  - Both implement data-parallel operators
  - Both can handle streaming data

- But there are key differences that impact use in live systems:

<table>
<thead>
<tr>
<th></th>
<th>IMDGs</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data characteristics</td>
<td>“Live” key-value pairs</td>
<td>Static data or batched streams</td>
</tr>
<tr>
<td>Data model for APIs</td>
<td>Object collections</td>
<td>Resilient distributed datasets</td>
</tr>
<tr>
<td>Focus of APIs</td>
<td>CRUD, query, data-structure methods</td>
<td>Data-parallel operators on RDDs for analytics</td>
</tr>
<tr>
<td>High availability</td>
<td>Data replication for fast recovery</td>
<td>Lineage for max performance</td>
</tr>
<tr>
<td>tradeoffs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IMDG’S DATA ACCESS MODEL

- Object-oriented APIs store unstructured collections of objects by type:
  - Accessible by key (string or number) using “CRUD” APIs
  - APIs in Java, C#, C++, REST, …
  - Also accessible by distributed query on properties
  - Stored within IMDG as serialized blobs or data structures

- Objects often have extended semantics:
  - Synchronization with distributed locking
  - Bulk loading
  - Timeouts, eviction
  - Transparent access to a backing store or remote IMDG
  - Methods on data structures

Basic “CRUD” APIs:
- Create(key, obj, tout)
- Read(key)
- Update(key, obj)
- Delete(key)
  and...
- Lock(key)
- Unlock(key)
static void Main(string argv[])
{
    // Initialize string object to be stored:
    String s = "Test string";

    // Create a cache collection:
    SossCache cache = SossCacheFactory.getCache("MyCache");

    // Store object in the grid:
    CachedObjectId id = new CachedObjectId(UUID.randomUUID());
    cache.put(id, s);

    // Read object stored in the grid:
    String answerJNC = (String)cache.get(id);

    // Remove object from the grid:
    cache.remove(id);
}
static void Main(string[] args)
{
    // Initialize object to be stored:
    SampleClass sampleObj = new SampleClass();
    sampleObj.var1 = "Hello, grid!";

    // Create a cache:
    SossCache cache = CacheFactory.GetCache("myCache");

    // Store object in the grid:
    cache["myObj"] = sampleObj;

    // Read object stored in grid:
    retrievedObj = cache["myObj"] as SampleClass;

    // Remove object from the grid:
    cache["myObj"] = null;
}
Use CRUD APIs to transparently access objects held within remote IMDGs.

Implements a virtual IMDG spanning multiple sites.

IMDG can implement coherency policies to mitigate WAN latency.
USING PARALLEL QUERY TO ACCESS DATA

- Retrieves *multiple* objects with selected properties and/or tags.
- Runs on all grid servers to scale query performance.
- Typically uses fast, indexed lookup on each grid server.
- Often used to read a set of related objects for analysis.
- **However, has O(N) overhead per query:**
  - Implies overall O(N**2) overhead with many clients
  - Can easily lead to network saturation
  - Not suitable as primary access method

Queries the IMDG in parallel.

Merges the keys into a list for the client.
Mark selected class properties as indexes for parallel query:

```java
public class Stock implements Serializable {
    private String ticker;
    private int totalShares;
    private double price;

    @SossIndexAttribute
    public String getTicker() {return ticker;} ...
}
```

Define query using these properties:

```java
NamedCache cache = CacheFactory.getCache("Stocks", false);
Set keys = cache.queryKeys(Stock.class,
    or(equal("ticker", "GOOG"),
    equal("ticker", "ORCL")));
```
PARALLEL QUERY EXAMPLE (C# WITH LINQ)

- Mark selected class properties as indexes for parallel query:

```csharp
public class Stock {
    [SossIndex]
    public string Ticker { get; set; }
    public decimal TotalShares { get; set; }
    public decimal Price { get; set; }
}
```

- Define query using these properties; optionally access objects automatically from IMDG:

```csharp
NamedCache cache = CacheFactory.GetCache("Stocks");
var q = from s in cache.QueryObjects<Stock>()
       where s.Ticker == "GOOG" || s.Ticker == "ORCL"
       select s;

Console.WriteLine("{0} Stocks found", q.Count());
```
RUNNING A METHOD IN THE CLIENT

- Client reads or queries objects, run a method, and then optionally update the grid.
- Data motion can place a huge burden on the network.
- This approach fails to make use of untapped computing power in IMDG hosts.
Example: E-commerce inventory management:

- E-commerce site stores orders and inventory changes within IMDG.
- Goal: reconcile orders and inventory by SKU in real time for 50K SKUs.
- Implementation using parallel query (pseudo-code):

```plaintext
parallel foreach (sku)
{
    orders[] = QueryIMDG(orders_ns, sku);
    stockch[] = QueryIMDG(stock_ns, sku);
    Reconcile(orders, stockch);
    SendAlerts();
}
```

- Problem: Queries quickly saturate the network.
THE EFFECT OF DATA MOTION

- Network access creates a bottleneck that limits throughput and increases latency.
- Avoiding data motion enables linear scalability for growing workloads => predictable, low latency.
- Example: back-testing stock histories in parallel
- How? Move the computation into the grid to avoid data motion.
BUILDING DATA STRUCTURES IN THE GRID SERVICE
 Client invokes method and ships parameters to an IMDG object.
- Parameters are typically much smaller than the object.
- IMDG runs the method in the local grid service and returns a result to the client.
- **Data motion is dramatically reduced.**
IMDG becomes a **data structure store** that implements useful data types and methods.

- Example: Redis (which provides lists, sets of strings, hashed sets, sorted sets)
- Advantages: fast (reduced data motion, no deserialization), secure, less client development
Stores an ordered sequence of string values within a single Redis object.

Implements client APIs for managing the list.

Allows Redis server to move items between lists.

Client avoids reading and updating entire list object to perform list operations.

**LIST:**

- `RPUSH` key-name value [value]
- `LPUSH` key-name value [value]
- `RPOP` key-name returns value
- `LPOP` key-name returns value
- `LRANGE` key-name start end
- `LTRIM` key-name start end
- `BLPOP` key-name [key-name ...]
- `BRPOP` key-name [key-name ...]
- `RPOPLPUSH` src-key dst-key
- `BRPOPLPUSH` src-key dst-key
BUILDING EXTENSIBLE DATA STRUCTURES IN REDIS

Users can extend data structures using Lua server-side scripts:

- Operate on local server objects.
- Run without interruption to implement atomic operations (avoiding multiple round trips).
- Call standard Redis commands within the script.
- Use to extend existing data structures or implement new ones (e.g., sharded list).

Breaking news: Upcoming Redis release will have loadable modules for greater extensibility.

```python
>>> r = redis.StrictRedis()
>>> lua = ""
... local value = redis.call('GET', KEYS[1])
... value = tonumber(value)
... return value * ARGV[1]"
>>> multiply = r.register_script(lua)
>>> r.set('foo', 2)
>>> multiply(keys=["foo"], args=[5])
10
```

Example of using a Lua script (Python) from https://pypi.python.org/pypi/redis
ISSUES WITH RUNNING IN THE GRID SERVICE

- **Performance:**
  - Running complex methods can slow down grid service’s data access.
  - A hot object can consume CPU and delay other accesses and methods.

- **Extensibility:**
  - Local server scripting does not support distributed data structures.
  - Scripting engine must be compatible with service's language runtime environment.

- **Reliability/Security:**
  - Extensible server-based scripting can crash or compromise the server.

**Alternative: run data structure methods in a separate worker process.**
- Solves extensibility issues: security, language flexibility, distributed access.
- But...creates tradeoffs: performance, complexity.
BUILDING EXTENSIBLE DATA STRUCTURES OUTSIDE THE GRID SERVICE
On each IMDG host, create a worker process which executes data-structure methods.

The set of worker processes is called an invocation grid.

IGs usually run language-specific execution environments (JVM, .NET runtime).

IGs handle execution requests from the local grid service.

Data-structure methods can access all objects in the IMDG.
Execution steps for running a method:

- Client invokes method on a specific object.
- Client library maps request to a grid host.
- Grid service forwards request to local worker process.
- Worker process executes method:
  - Grid service forwards serialized object and parameters to the worker; to mitigate overhead:
    - Worker maintains a client-side cache of recently accessed objects.
    - Worker can use memory-mapped file.
  - Method optionally returns result to grid service.
  - Grid service optionally returns result to client.
ONE TECHNIQUE: “SINGLE METHOD INVOCATION”

Leverages Object-Oriented View of Grid Data:

- Assume that an IMDG collection (“name space”) contains objects of a single type.
- Each object encapsulates a data structure managed by user-defined code.
- SMI invokes user-defined method on a specified instance within the name space using object’s key:
  - Optionally passes in a parameter object.
  - Optionally returns a result object.

```c
result = namespace.SingleObjectInvoke(key, SmiMethod, param_obj);
```

- Can encapsulate SMI call into user-defined API wrapper to simplify client’s view:

```c
ResultType ClientMethod(GridKey key, ParamType param1, ...) {
    <setup namespace>;
    <copy parameters into params_obj>;
    return namespace.SingleObjectInvoke(key, SmiMethod, param_obj);
}
```
EXAMPLE: REDIS LIST DATA STRUCTURE USING SMI (C#)

- Create server-side data structure and methods:

```csharp
public class StackList<T> : LinkedList<T> {
    public StackList() : base() {}
    public static void LPush(StackList<T> list, T item) { list.AddFirst(item); }
    public static void RPush(StackList<T> list, T item) { list.AddLast(item); }
    public static T LPop(StackList<T> list) {
        T retItem = (list.First as LinkedListNode<T>).Value;
        base.RemoveFirst(); return retItem;
    }
    public static T RPop(StackList<T> list) {
        T retItem = (list.Last as LinkedListNode<T>).Value;
        base.RemoveLast(); return retItem;
    }
}
```

- Run client wrapper which invokes server-side method on target host (pseudo-code here):

```csharp
InvokeLPush<T>(GridKey list-key, LinkedListNode<T> item) {
    return nc.SingleObjectInvoke(list-key, LPush, item);
}
```
public class StockPriceData {
    private double stockPrice;
    private double peRatio; ...
};

StockPriceData[] data = new StockPriceData[]{
    new StockPriceData(100.3, 10.4),
    new StockPriceData(102.1, 20.1), ...
};

NamedCache pset = CacheFactory.getCache("stockCache");
pset.add("stockData", data);

public class StockInvokable implements Invokable<StockPriceData[],
    Integer, Double> {
    public Double eval(StockPriceData[] data, Integer param){
        <analyze data and return Double result> }
};

Double result = pset.singleObjectInvoke(new StockInvokable(),"stockData",data.length);
First obtain a reference to the IMDG's object collection of portfolios:

```java
NamedCache pset = CacheFactory.getCache("portfolios");
```

Create an “invocation grid;” a re-usable compute engine for the application:

- Spawns a JVM on all grid servers and connects them to the in-memory data grid.
- Stages the application code and dependencies on all JVMs.
- Associates the invocation grid with an object collection.

```java
InvocationGrid grid = new InvocationGridBuilder("grid")
    .addClass(DependencyClass.class)
    .addJar("/path/to/dependency.jar")
    .setJVMParameters("-Xmx2m")
    .load();

pset.setInvocationGrid(grid);
```
IMDG creates IG worker processes in parallel

1. Start IG
2. Multicast all hosts to start IG
3. Read code object into IG workers
Concurrent method execution provides parallel speedup:

- IMDG handles multiple streaming requests from a single client.
- Also handles multiple clients in parallel.
- Provides “embarrassingly parallel” speedup.
Pros and Cons of Using an Invocation Grid

Advantages:
- Secure:
  - User code cannot access grid data.
  - User code cannot bring down grid service.
- Flexible:
  - Supports arbitrary languages.
  - Supports full IMDG APIs and access to remote objects.
  - Enables building distributed data structures.

Disadvantages:
- Higher latency due to round trip from grid service to IG worker:
  - Mitigated by client cache in IG worker and memory-mapped files
  - Additional CPU and data motion on grid hosts
  - Complexity of implementing code shipping and IG worker orchestration
THE NEXT STEP: DATA-PARALLEL DATA STRUCTURES AND METHODS
WHY USE DATA-PARALLEL METHODS?

Leverages the grid’s scalability to keep latency low for large workloads.

- Limitations of single method execution:
  - Data structures must fit within a single object.
  - Does not leverage scalability of the IMDG to perform a single operation on many objects.

- Data-parallel method execution:
  - Implements a data structure operation spanning multiple objects.
  - Examples: MapReduce, weather simulation
  - Runs computation in parallel on all objects and optionally combines results.

- Benefit: handles very large data structures with low latency and scalable speedup.
DATA-PARALLEL METHOD EXECUTION

- Client runs a method on multiple objects distributed across all grid hosts.
- Objects are usually part of one typed collection.
- Results are merged into a single result returned to the client.
- Provides a building block for complex computations (e.g., MapReduce).
Like SMI, PMI leverages object-oriented view of grid data:

- Follows standard HPC model (broadcast/merge) for data-parallel computation.
- Assumes that an IMDG collection (name space) contains objects of a single type.
- Each object maintains a data structure managed by user-defined code.
- PMI invokes user-defined “eval” method in parallel on a set of queried instances within the name space:
  - Optionally passes in a parameter object.
  - Optionally returns a result object.
- PMI invokes optional user-defined “merge” method in parallel to perform binary merging across all results:
  - Returns a single result to the client (or serves as a barrier).
  - Uses binary merge tree to combine all results in log-2(N) time where N is # grid hosts.

```csharp
result = namespace.ParallelInvoke(query_spec, Eval, Merge, param_obj);
```
EXAMPLE IN FINANCIAL SERVICES

- **Goal**: track market price fluctuations for a hedge fund and keep portfolios in balance.

- **How**:
  - Keep portfolios of stocks (long and short positions) in object collection within IMDG.
  - Collect market price changes in one-second snapshots.
  - Define a method which applies a snapshot to a portfolio and optionally generates an alert to rebalance.
  - Perform repeated parallel method invocations on a selected (i.e., queried) set of portfolios.
  - Combine alerts in parallel using a user-defined merge method.
  - Report alerts to UI every second for fund manager.
UI FOR FINANCIAL SERVICES EXAMPLE

- UI performs a PMI every second with a new market snapshot.
- User can select filters to parameterize PMI.
- PMI completes in ~350 msec.
- UI lists all portfolios (on left).
- UI highlights alerted portfolios from PMI merge.
- User selects and reads portfolio object within grid to see alerted positions (on right).
PORTFOLIO EXAMPLE: DEFINING THE DATASET

- Simplified example of a portfolio class (Java):
  - Note: some properties are made query-able.
  - Note: the `evalPositions` method analyzes the portfolio for a market snapshot.

```java
public class Portfolio {
    private long id;
    private Set<Stock> longPositions;
    private Set<Stock> shortPositions;
    private double totalValue;
    private Region region;
    private boolean alerted; // alert for trading
    @SossIndexAttribute // query-able property
    public double getTotalValue() {...}
    @SossIndexAttribute // query-able property
    public Region getRegion() {...}

    public Set<Long> evalPositions(MarketSnapshot ms) {...};
}
```
DEFINING THE PARALLEL METHODS

- Implement interface that defines methods for analyzing each object and for merging the results:

```java
public class PortfolioAnalysis implements Invokable<Portfolio, MarketSnapshot, Set<Long>> {
    public Set<Long> eval(Portfolio p, MarketSnapshot ms) throws InvokeException {
        // update portfolio and return id if alerted:
        return p.evalPositions(ms);
    }

    public Set<Long> merge(Set<Long> set1, Set<Long> set2) throws InvokeException {
        set1.addAll(set2);
        return set1; // merged set of alerted portfolio ids
    }
}
```
INVOKING THE PARALLEL METHOD INVOCATION

- Run the PMI on a queried set of objects within the collection:
  - Multicasts the invocation and parameters to all JVMs.
  - Runs the data-parallel computation.
  - Merges the results and returns a final result to the point of call.

```java
InvokeResult alertedPortfolios = pset.invoke(
    PortfolioAnalysis.class,
    Portfolio.class,
    and(greaterThan("totalValue", 1000000), // query spec
      equals("region", Region.US)),
    marketSnapshot, // parameters
    ...);

System.out.println("The alerted portfolios are" +
                   alertedPortfolios.getResult());
```
**EXECUTION STEPS**

- **Eval phase**: each server queries local objects and runs eval and merge methods:
  - Note: accessing local data avoids networking overhead.
  - Completes with one result object per server.

- **Merge phase**: all servers perform distributed merge to create final result:
  - Merge runs in parallel to minimize completion time.
  - Returns final result object to client.
THE GOAL: SCALABLE SPEEDUP

- Assume that a workload of size $W$ requires time $T$ to complete on a single server:
  - Throughput on 1 server = $1/T$
  - Latency on 1 server = $T$
- Non-goal: Use $N$ servers to decrease latency to $T/N$ for fixed-size workload $W$.
- Goal: linearly scale throughput while workload grows:
  - Workload grows $N$-fold on $N$ servers.
  - **Throughput on $N$ servers = $N/T$ for workload of size $N*W$.**
  - **Latency on $N$ servers = $T$ (does not grow).**
  - Example: web farm from handling $W$ users to $N*W$ users.
- Bottlenecks to speedup lower throughput and increase latency:
  - Usually due to network congestion.
  - Can be due to other resource contention (e.g., hot objects).
IMDG DELIVERS SCALABLE SPEEDUP

Example: Measured a similar financial services application (back testing stock trading strategies on stock histories):

- Hosted IMDG in Amazon EC2 using 75 servers holding 1 TB of stock history data in memory
- IMDG handled a continuous stream of updates (1.1 GB/s)
- Results: analyzed 1 TB in 4.1 seconds (250 GB/s).
- Observed near linear scaling as dataset and update rate grew.
Sharding allows concurrent (and data-parallel) operations and enables scalable speedup.

- **Step 1: Sharding within a single server:**
  - Splits a single object into multiple parts (“shards”).
  - Allows concurrent operations on different shards.
  - Takes advantage of multiple CPU cores within a server for parallel speedup.

- **Example: sharding a Redis list (from Redis in Action)**
  - Uses Lua server-side script.
  - Splits list into shard objects.
  - Uses well-known local objects to find list ends.
  - Limitation: cannot distribute shards across servers.
SCALING ACROSS MULTIPLE SERVERS

- **Step 2: Sharding across multiple servers:**
  - Handles N-times larger workload (memory use) on N servers.
  - Enables linearly increasing throughput and maintains low latency as workload grows.

- **One technique: shard in client:**
  - Clients map shards to servers.
  - Servers manage shards independently (“embarrassingly parallel”).
  - Example: distributed Redis hashed set
  - Issue: how to re-shard when servers are added or removed?
EXAMPLE: JAVA CONCURRENT MAP

- Challenge: How to store very large number of small key-value pairs within an IMDG:
  - Access KVPs individually using Java concurrent map semantics.
  - Access/update KVPs in parallel using a MapReduce engine for analysis.

- One implementation (ScaleOut “Named Map”):
  - Shard chunks of KVPs across IMDG servers.
  - Use hash table within each chunk for fast lookup.
  - Pipeline chunks to/from MapReduce engine as splits/partitions.
  - Host chunks as normal IMDG objects (with server-based sharding) for automatic load-balancing.
COMBINING SINGLE AND DATA-PARALLEL OPERATIONS ON DATA STRUCTURES
COMBINING SMI & PMI IN ONE APPLICATION

- Example: Tracking cable TV set-top boxes
- Goals:
  - Make real-time, personalized upsell offers
  - Detect and manage service issues
  - Track aggregate behavior to identify patterns, e.g.:
    - Total instantaneous incoming event rate, network hot spots
    - Most popular programs and # viewers by zip code
- How:
  - Represent set-top-boxes as data structure objects held in the IMDG.
  - Use **SMI** to track events from 10M set-top boxes with 25K events/sec (2.2B/day):
    - Correlate, cleanse, and enrich events per rules (e.g. ignore fast channel switches, match channels to programs)
    - Feed enriched events to recommendation engine within 5 seconds
  - Use **PMI** with a named map to track aggregate statistics across all set-top boxes every 10 seconds.
Each set-top box is represented as an object in the in-memory data grid (IMDG).

Object holds raw & enriched event streams, viewer parameters, and statistics

IMDG captures incoming events and correlates/cleanses/enriches using SMI:
- Note: IMDG makes an excellent stream-processing engine.
- Avoids data motion of other approaches.

IMDG uses PMI across all set-top box objects to periodically collect and report global statistics.
PERFORMING CROSS-SERVER SMI

- SMI may need to invoke a second method on a remote object.
- Example: enriching channel-change events from a program guide
- Requirements:
  - SMI must have global view of IMDG objects.
  - SMI must be able to allow asynchronous execution of a remote request.
- Solution:
  - Run method within IG worker (vs. grid service).
  - Enable fully distributed view of IMDG in SMI.
Example from a proof of concept UI based on a simulated workload for San Diego metropolitan area:

- Runs on a ten-server cluster.
- Continuously correlates and cleanses telemetry from 10M simulated set-top boxes (from synthetic load generator).
- Processes more than 30K events/second.
- Enriches events with program information every second.
- Tracks and reports aggregate statistics (e.g., top 10 programs by zip code) every 10 seconds.

**THE RESULT: REAL-TIME STREAMING & STATISTICS**

Real-Time Dashboard
RECAP: ADDING DATA STRUCTURES TO IMDGS

- Built-in data structures (e.g., Redis) provide a pre-defined set of fast, easy to use data structures.

- Moving computation into an in-memory data grid offers several benefits:
  - Reduces network bottlenecks due to data motion.
  - Leverages CPU power of the IMDG’s hosts.
  - Enables scalable speedup to handle large workloads with low latency.

- Use of worker process increases security, reliability, and flexibility, although it increases overhead.

- Extensible, grid-based data structures enable efficient implementation of complex computations.
  - These techniques can be used for both single-object and data-parallel computations.

Bottom line: IMDGs are a great platform for in-memory computing.