Using In-Memory Computing to Create the Digital Twin: A New Model for Stream Processing

DR. WILLIAM L. BAIN
SCALEOUT SOFTWARE
A Brief Journey Towards the Digital Twin

- Quick Review of Stream Processing
- How In-Memory Data Grids Differ
- What is a Digital Twin?
- Implementing a Digital Twin with an IMDG
- Data Ingestion and Time Windows
- Implementing & Using Digital Twins
About the Speaker

• Dr. William Bain, Founder & CEO of ScaleOut Software:
  • Email: wbain@scaleoutsoftware.com
  • 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 for:
  • Scaling application performance with in-memory data storage
  • Analyzing live data in real time with in-memory computing

• Twelve years in the market; 440+ customers, 11,000+ servers
Basic Stream-Oriented Architecture

Stream-oriented platforms typically create a computing pipeline from data sources to sinks:

- Pipeline stages perform transformations often described by programming models as a sequence of extension methods.
- Usually access state data (in-memory and/or persistent) using an optional, separate storage tier.
- Examples: Apama (CEP), Apache Storm, Spark Streaming, Beam, and Flink
Complex Event Processing Architecture

• Example: Apama from Software AG

• Architecture (the Apama “Correlator”):
  • HyperTree: matches and filters incoming events
  • Temporal Sequencer: finds real-time correlations between events
  • Stream Processor: executes analytics on windows of events

• Programs can be written in EPL or Java; simple example of stock tracking in EPL:

```epl
monitor PriceRise{
    StockTick firstTick, finalTick;
    action onload() {
        on StockTick (symbol="IBM", price>210.5):firstTick {furtherRise();}
        action furtherRise() on StockTick (symbol="IBM",
            price>firstTick.price*1.05):finalTick {
            send PlaceSellOrder("IBM", 100.0 to "Market");
        }
    }
}
```

Illustration and code sample from “The Apama Platform,” Software AG
Two Apache Platforms for Stream Processing

Task-parallel:

Data-parallel (micro-batched):
Stream Processing Model from Apache Beam

- Originally developed by Google.
- Provides unified, portable APIs for batch and stream processing.
- Relies on external execution platforms called “runners” (e.g., Apache Flink, Spark, Google Cloud Dataflow).

Key elements:
- Pipeline: data processing job as a directed set of steps
- PCollection: the data inside a pipeline
- PTransform: an execution step in the pipeline (e.g., ParDo) or an IO step

Illustration from “Introduction to Apache Beam” by JB Onofre’
Apache Beam Code Examples (Java)

• Basic Dataflow model:

```java
Pipeline p = Pipeline.create(); // create a pipeline
p.apply(TextIO.Read.from("/path/to/…") // read input
    .apply(new CountWords()) // do some processing
    .apply(TextIO.Write.to("/path/to/…"); // write output
p.run(); // run the pipeline
```

• Example of advanced features (session windows):

```java
PCollection<KV<String, Integer>> scores = input
    .apply(Window.into(SessionWindows.of(Duration.standardMinutes(2))
        .triggering(AtWatermark()
            .withEarlyFirings(
                AtPeriod(Duration.standardMinutes(1)))
            .withLateFirings(AtCount(1)))
        .accumulatingFiredPanes())
    .apply(Sum.IntegersPerKey());
```
Stream Processing with Apache Flink

- Flink data flow:

- Flink architecture:

How In-Memory Data Grids Differ

IMDGs focus on integrating computing with state (vs. processing data streams with optional external state):

- IMDG provides scalable, hi-av storage for live data:
  - Stores and manages live state with object-oriented model:
    - Sequentially consistent data shared by multiple clients
    - Object-oriented collections by type
    - CRUD APIs for data access as key/value pairs
    - Distributed query by object properties
  - Has fast (<1 msec.) data access and updates
  - Designed for transparent scalability and high availability:
    - Automatic elasticity and load-balancing
    - Automatic data replication, failure detection, recovery
- IMDG integrates in-memory computing with data storage:
  - Leverages the computing power of commodity servers.
  - Computes where the data lives to avoid network bottlenecks.
Adding In-Memory Computing to an IMDG

• Each grid host runs a worker process which executes application-defined methods.
  • The set of worker processes is called an invocation grid.
  • IG usually runs language-specific runtimes (JVM, .NET).
  • IMDG can ship code to the IG workers.

• Key advantages:
  • Avoids network bottlenecks by moving computing to the data.
  • Leverages IMDG’s cores & hosts.
  • Isolates application code from grid service.
IMDGs Perform Both Stream and Batch Processing

- IMDG leverages object-oriented storage model to execute methods on instances of stored objects.
- IMDG naturally integrates both stream-based and batch execution models:
  - Stream-based: execute method(s) on independent objects and sequentially on the same object.
  - Result: an implementation of the HTAP architecture
Example of Combining Streaming and Batch

An Ecommerce site tracking web shoppers:

- IMDG manages clickstreams from shoppers by calling methods on individual objects to process click events.
  - Can immediately track shopper’s actions.

- IMDG performs data-parallel, batch analytics on grid data to track aggregate trends.
  - Can determine best selling products, average basket size, etc.
Executing Multiple, Independent Requests

Method execution runs independently for multiple objects:

• IMDG handles streaming requests from a single client.

• Also handles multiple clients in parallel.
Executing a Data-Parallel Method

Method execution implements a batch job on an object collection:

- Client runs a single method on multiple objects distributed across the grid.
- Results optionally are merged and returned to the client.
Basic Data-Parallel Execution Model

A fundamental model from parallel supercomputing:

• Run one method (“eval”) in parallel across many data items.

• Optionally **merge** the results.
  • Binary combining is a special case, but...
  • It runs in \( \log N \) time to enable scalable speedup.
MapReduce Builds on This Model

- Runs in two data-parallel phases (map, reduce):
  - **Map** phase repartitions and optionally combines source data.
  - **Reduce** phase analyzes each data partition.
  - A global merge of the results is not performed.

- Classic example: word count
  - Source data items: lines of text
  - Mappers: emit `{word, count}` for all unique words.
  - Words are hashed to partitions.
  - Reducers sum counts and emit total counts for each word.
Data-Parallel Execution Steps

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

- **Merge** phase: all servers perform binary, distributed merge to create final result:
  - Merge runs in parallel to minimize completion time.
  - Returns final result object to client.
Ecommerce Code Sample (C#)

• Define shopping cart objects stored in the in-memory data grid (IMDG):

```csharp
class ShoppingCartItem
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public int Quantity { get; set; }
}

class ShoppingCart
{
    public string CustomerId { get; set; }
    public IList<ShoppingCartItem> Items { get; } = new List<ShoppingCartItem>();
    public decimal TotalValue
    {
        get { return Items.Sum((item) => item.Quantity * item.Price); }
    }
    public decimal ItemCount
    {
        get { return Items.Sum((item) => item.Quantity); }
    }
}
```
Loading the Shopping Carts into the Grid

- IMDG provides location-independent access using create/read/update/delete ("CRUD") APIs.

```csharp
var carts = CacheFactory.GetCache("carts"); // Gets reference to a namespace
foreach (var cart in collection)
    carts.Add(cart.CustomerId, cart); // CustomerId serves as key
```

- IMDG transparently distributes and load-balances the shopping carts across a cluster of servers or cloud instances.

- Allows an application to host much larger data sets than possible on a single server.
private static void PostCartItem()
{
    var nc = CacheFactory.GetCache("carts");
    var item = new ShoppingCartItem
    {
        Name = "Acme Snow Globe",
        Price = 7.50m,
        Quantity = 3
    };
    var key = nc.CreateKey("Jane Doe");
    nc.PostEvent(id: key,
        eventInfo: "Add cart item",
        payload: item.ToBytes());
}
Running a Streaming Method on a Single Object

// Initialization method is run when the invocation grid is first loaded:
public void Init_pipeline()
{
    // Set up a ReactiveX pipeline to handle adding shopping cart items:
    carts.GetEventSource()
        .Where(ev => ev.EventInfo == "Add cart item")
        .Select(ev => Tuple.Create(ShoppingCartItem.FromBytes(ev.Payload),
                                  ev.ObjectId.GetStringKey()))
        .Subscribe(HandleCartAddEvent);
}

public void HandleCartAddEvent(Tuple<ShoppingCartItem, string> addCartItemTuple)
{
    var custId = addCartItemTuple.Item2;
    var mycart = carts.Retrieve(custId, acquireLock: true) as ShoppingCart;
    mycart.Items.Add(addCartItemTuple.Item1);
    carts.Update(custId, mycart, unlockAfterUpdate: true);
}
Running a Batch Data-Parallel Method

```csharp
finalResult = carts.QueryObjects<ShoppingCart>().Where(cart => cart.TotalValue >= 20.00m) // filter carts .Invoke(
    timeout: TimeSpan.FromMinutes(1), param: productName,
    evalMethod: (cart, pName) =>
    {
        var result = new Result();
        result.numCarts = 1;
        // see if the selected product is in the cart:
        if (cart.Items.Any(item => item.Name.Equals(pName)))
            result.numMatches++;
        return result;
    })
    .Merge(
       (result1, result2) =>
       {
           result1.numMatches += result2.numMatches;
           result1.numCarts += result2.numCarts;
           return result1; });
```
What Is a Digital Twin?

- Term coined by Dr. Michael Grieves (U. Michigan) in 2002 for use in product life cycle management
- Popularized in Gartner’s “Top 10 Strategic Technology Trends for 2017: Digital Twins” for use with IoT
- Definition: a digital representation of a physical entity; an encapsulated software object that comprises (per Gartner):
  - A model (e.g., composition, structure, metadata for an IoT sensor)
  - Data (e.g., sensor data, entity description)
  - Unique identity (e.g., sensor identifier)
  - Monitoring (e.g., alerts)

- Significance: focuses on modeling data sources
  - A basis for correlating and analyzing streaming data
  - A context for deep introspection and interaction
Examples of Digital Twins in IoT

**Live System - Physical Objects**
- (Autonomous) Vehicles
- Manufacturing floors and equipment
- Wind turbines and wind farms

**Digital Twins**
- Vehicle subsystems for safety monitoring & predictive maintenance
- Networks of machine tooling for real-time interactive view and predictive maintenance
- Collections of wind turbine components for remote operations and predictive maintenance

Telemetry streams
Immediate feedback
Tracking an Elevator: A Digital Twin Demonstration

Digital twin of an elevator implemented by Crossvale, Inc.:

Real-World Elevator

**Elevator Specs**
- Max People: 10
- Max Weight: 1000 kg
- Floors: 8

**Current Load**
- People: 4
- Weight: 320 kg

**Actions**
- Open Door
- Close Door
- Ascend
- Descend
- Return to Lobby

**Telemetry Streams**

**Monitoring Stats**
- Elevator Position (cm): 720 cm
- Floor position: 3
- Onboard Weight (kg): 320 kg
- Power Consumption (kW): High

**Events**
- Operation Normal
- Learning
- Overweight
- Descend Too-Fast
- Ascend Too-Slow
- Stuck Between Floors
- Doors Stuck

**Alerts**
- Check Cable
- Vibration Limit Exceeded
- Maintenance Required

Courtesy of: Crossvale
Some Applications for Digital Twins

A digital twin integrates incoming events with state information using domain-specific algorithms to generate alerts:

<table>
<thead>
<tr>
<th>Application</th>
<th>State Information</th>
<th>Events</th>
<th>Logic</th>
<th>Alerts</th>
</tr>
</thead>
<tbody>
<tr>
<td>IoT devices</td>
<td>Device status &amp; history</td>
<td>Device telemetry</td>
<td>Analyze to predict maintenance.</td>
<td>Maintenance requests</td>
</tr>
<tr>
<td>Medical monitoring</td>
<td>Patient history &amp; medications</td>
<td>Heart-rate, blood-pressure, etc.</td>
<td>Evaluate measurements over time windows with rules engine.</td>
<td>Alerts to patient &amp; physician</td>
</tr>
<tr>
<td>Cable TV</td>
<td>Viewer preferences &amp; history, set-top box status</td>
<td>Channel change events, telemetry</td>
<td>Cleanse &amp; map channel events for reco. engine; predict box failure.</td>
<td>Viewer recommendations, repair alerts</td>
</tr>
<tr>
<td>Ecommerce</td>
<td>Shopper preferences &amp; buying history</td>
<td>Clickstream events from web site</td>
<td>Use ML to make product recommendations.</td>
<td>Product list for web site</td>
</tr>
<tr>
<td>Fraud detection</td>
<td>Customer status &amp; history</td>
<td>Transactions</td>
<td>Analyze patterns to identify probable fraud.</td>
<td>Alerts to customer &amp; bank</td>
</tr>
</tbody>
</table>
Why Use an IMDG to Host Digital Twins?

- **Object-oriented data storage:**
  - Offers a natural model for hosting digital twins.
  - Cleanly separates domain logic from data-parallel orchestration.
  - Provides rich context for processing streaming data.
  - Integrates streaming and batch processing.

- **High performance:**
  - Avoids data motion and associated network bottlenecks.
  - Fast and scales to handle large workloads.

- **Integrated high availability:**
  - Uses data replication designed for live systems.
  - Can ensure that computation is high av.
Modeling the Digital Twin with OOP

• Digital twin typically comprises:
  • An event collection
  • State information about the data source
  • Logic for managing events, updating and analyzing state, generating alerts

• Object oriented model:
  • Integrates event collection with state information.
  • Encapsulates domain-specific logic (e.g., ML, rules engine, etc.).
  • Runs code where the data lives (avoids data motion).
  • Delivers fast response times.
Comparison to Stream-Oriented Platforms

Stream-oriented platforms typically focus on analyzing the event stream:

- Lack specific support for building digital twins and managing their state & semantics:
  - Adds complexity in implementing digital twin models.
  - Can lack a clean separation between event orchestration and domain-specific code.

- Do not specifically integrate state management with stream processing:
  - Usually require state data to be accessed or updated using a separate storage tier.
  - Incur network delays which can lead to bottlenecks.
The Effect of Data Motion on Scaling

- Data motion creates a bottleneck that limits throughput.
- Avoiding data motion enables linear scalability for growing workloads => predictable, low latency.
- Example: back-testing stock histories in parallel
Comparison to Stream-Oriented Platforms

Some advantages of the digital twin model:

- Auto-correlates events from each data source:
  - Avoids the need to do this in the stream processing pipeline.

- Refactors processing steps to perform them in one location:
  - Avoids possible data motion between steps.

- Provides a basis for transparent scaling:
  - Leverages the grid’s load-balancing of digital twin objects across the IMDG.
Ingesting Stream Data into an IMDG from Kafka

IMDG can transparently scale event reception from Kafka:

• IMDG can spawn multiple Kafka connectors in a “Connector Grid” to handle events in parallel.
• IMDG can spawn a “Worker Grid” to receive events and implement digital twin semantics.
• IMDG transparently scales as the workload grows.

IMDG uses key to direct events to grid host for associated digital twin object.
Code Sample (Java): Connecting an IMDG to Kafka

```
// Create a grid startup action to start Kafka connectors:
GridAction connectAction = new ConnectorGridBuilder("hr_cache")
    .addKafkaServerPropertiesPath(new File("server.properties"))
    .addConnectorProperties(new File[] {new File("sink.properties"),})
    .build();

// Start the invocation grid and register the startup action:
InvocationGrid grid = new InvocationGridBuilder("conn_grid")
    .setLibraryPath("Kafka").addJar("applicationClasses.jar")
    .addStartupAction(connectAction).load();

# Example of connect-grid-sink.properties:
name=grid-sink
connector.class=GridSinkConnector
key.converter=PassThroughConverter
value.converter=PassThroughConverter
topics=my_kafka_topic
grid.namedcache.name=mycache
```
Using Kafka Partitions to Scale Event Handling

• Kafka offers partitions to scale out handling of event messages.
  • Partitions are distributed across brokers.
  • Brokers process messages in parallel.

• IMDG can map Kafka partitions to grid partitions.

• This minimizes event handling latency.
  • Avoids store-and-forward within IMDG.

• How?
  • IMDG specifies key mapping algorithm.
  • Application specifies # Kafka partitions.
  • IMDG listens to appropriate Kafka partitions (and handles membership changes).
Digital Twin Manages Time Windows of Events

• Each digital twin object can host a time-ordered & windowed collection of events.
  • Can be implemented as a transform on the collection similar to streaming APIs (e.g. Beam)

• Event posting triggers eviction based on windowing policy.

• Time window manager implements multiple windowing policies, e.g.:
  • Sliding
  • Tumbling
  • Session

• Time window manager implements queries that supply windowed events for analysis.
Example: A Heart-Rate Monitoring Application

A simple medical application that monitors heart rate telemetry from a mobile device:

• Receives heart-rate telemetry events from patient’s mobile device.
• Digital twin holds telemetry and patient’s history/status.
• Event posting logic tracks these events within a collection in the digital twin.
• Analysis logic evaluates the events using time windows on the collection and with regard to the patient’s history and status.
• In this example, it alerts a doctor when heart-rate exceeds age-specific threshold.
• Updates the patient’s status.
Medical Monitoring & Alerting Architecture

- Heart-rate events flow to their respective digital twin objects for processing.
- The IMDG transparently scales to handle large numbers of patients.
Code Sample (C#): Heart-Rate Monitor

// Heart-rate event:
public class HeartRate
{
    public string PatientID { get; set; }
    public DateTime Timestamp { get; set; }
    public short BeatsPerMin { get; set; }
}

// Patient (the digital twin):
public class Patient
{
    public string Id { get; set; }
    public IList<HeartRate> HeartRates { get; set; }
    public DateTime Birthdate { get; set; }
    public int Age => (int)Math.Floor((DateTime.Now - Birthdate).TotalDays / 365);
    public bool HeartIssueDetected { get; set; }
}
// Set up a ReactiveX pipeline in the IMDG to handle incoming heart-rate events:
heartMonGrid.GetEventSource()
    .Where(ev => ev.EventInfo == "Heart Rate Event")  // look for heart-rate events
    .Select(ev => HeartRate.FromBytes(ev.Payload))    // extract heart-rate data
    .Subscribe(HandleHeartRateEvent);                // update digital-twin
Code Sample (C#): Heart-Rate Monitor

```csharp
// Process an incoming heart rate event in the digital twin:
static void HandleHeartRateEvent(HeartRate heartRateEvent)
{
    var patient = heartMonGrid.Retrieve(heartRateEvent.PatientID,
        acquireLock: true)
        as Patient;

    // Obtain an enumerable windowing transformation of the event collection:
    var slidingHeartRates = new SlidingWindowTransform<HeartRate>(

    slidingHeartRates.Add(heartRateEvent); // add event and evict as necessary

    AnalyzePatient(patient, slidingHeartRates); // analyze & update patient’s status
}
```
// Analyze patient’s state and send an alert if necessary:
static void AnalyzePatient(Patient patient,
    SlidingWindowTransform<HeartRate> slidingHeartRates)
{
    // See if there are any 5-minute periods in the past day when the average
    // heart rate is too high. We use the sliding windows to calculate a
    // moving average and vary the alert threshold depending on patient’s age:

    foreach (var window in slidingHeartRates)
    {
        if (window.Count == 0) continue; // can't average zero elements

        var avg = window.Average(hr => hr.BeatsPerMin);
        if ((patient.Age > 50 && avg > 130) || avg > 160) {
            SendAlert($"{patient.Id} registers high heart rate at
            {window.StartTime}");
            patient.HeartIssueDetected = true;
        }
    }
}
A More Sophisticated Digital Twin Model

Example Model of Heart-Rate Monitoring for High Intensity Exercise Program

• Example of data to be tracked:
  • Event collection: time-stamped heart rate telemetry, type of exercise, specific parameters (distance, strides, altitude change, etc.)
  • Participant background/history: age, height, weight history, heart-related medical conditions and medications, injuries, previous medical events
  • Exercise tracking: session history, average # sessions per week, average peak heart rates, frequency of exercise types
  • Aggregate statistics: average/max/min exercise tracking statistics for all participants

• Example of logic to be performed:
  • Notify participant if session history across time windows indicates need to change mix.
  • Notify participant if heart rate trends deviate significantly from aggregate statistics.
  • Alert participant/medical personnel if heart rate analysis across time windows indicates an imminent threat to health.
  • Report aggregate statistics.
Challenge: Edge vs. Grid Service

How partition the digital twin model’s data and logic between edge devices and the grid service:

• Edge device:
  • Has limited storage and computing power, but...
  • Offers lowest latency to process events.

• Grid service:
  • Can run sophisticated algorithms.
  • Can store long event history.
  • Can track detailed state of the physical twin.

• Approach (akin to nervous system):
  • Perform tactical processing at edge for fast responsiveness.
  • Perform strategic processing in grid service.

• Software tools are needed for transparent migration.
Real-World Example: Tracking Cable Viewers

- **Cable Company’s Goals:**
  - Make real-time, personalized upsell offers.
  - Immediately respond to service issues & hotspots.
  - Track aggregate behavior to identify patterns, e.g.:
    - Total instantaneous incoming event rate
    - Most popular programs and # viewers by zip code

- **Requirements:**
  - 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) within 5 seconds.
  - Refresh aggregate statistics every 10 seconds.
Example: Tracking Cable Viewers

Solution:

• Each **set-top box** is represented as a digital twin object in the IMDG.
  • Holds raw & enriched event streams, viewer parameters, and box statistics.

• Use stream processing on box events to generate alerts for recommendation engine.

• Use periodic data-parallel operations on objects to generate aggregate statistics.

AWS Simulation:
• 25 servers
• 30K events/sec
• <1 sec. latency for alerts
• 10s per batch update
Example: Ecommerce Recommendations

• **Goals:**
  • Make real-time, personalized recommendations for an ecommerce web site:
    • Combine clickstream, shopper demographics, static recommendations
  • Track aggregate site performance, e.g.:
    • Shopper behavior (clicks-to-cart, basket size, ...)
    • Merchandizing effectiveness (best selling products)

• **Requirements:**
  • Handle 500K+ simultaneous shoppers.
  • Return recommendations within 200 msec.
  • Refresh aggregate statistics every minute.
Example: Ecommerce Recommendations

Solution:

• Each **shopper** is represented as a digital twin object in the IMDG.
  • Holds clickstream events, shopper demographics, and ML parameters.
  • Note: digital twins can be used to represent people.

• Use stream processing on clickstream events to generate recommendations.
  • Analysis logic runs an ML algorithm in real-time to generate recommendations.

• Use periodic data-parallel operations on objects to generate aggregate statistics.
Recap of the Journey

Traditional streaming focuses on stream data vs. data sources.

IMDGs use o-o state to integrate streaming & batch.

Digital twins enable deep introspection on stream data.

IMDG scales performance for digital twins.

OOP on an IMDG provides support for digital twin model.

Digital twins provide a powerful model for stream processing.

Thank you!
In-Memory Computing for Operational Intelligence

www.scaleoutsoftware.com