SAP Sailing Analytics

Architecture Overview

2012-08-03, Axel Uhl

# Introduction, Project Background and History

The SAP Sailing Analytics are a technology show-case demonstrating SAP technologies, concepts, skills and values applied to the domain of regatta sailing. They started as a small tool primarily intended to support a commentator in his job by displaying a live leaderboard for a sailing regatta with data interesting for the commentary. GPS and wind data travel from sensors to the server where the application keeps it in memory. When a request for a leaderboard is received, the data is aggregated on the fly, performing geometric computations including wind projections and involving a virtual "advantage line" orthogonal to the wind direction.

The live leaderboard started as a web application with a Java back-end responsible for the connectivity with the sensors and providing the geometry engine, and a Python process rendering the Web UI for the client's browser. The Python process issued REST requests to the Java back-end which responded with JSON documents.

The solution was first shown at Kieler Woche 2011. At the time, it was capable of displaying a single leaderboard that showed a number of tracked races in numerical form, offering columns for overall rank, race rank, rank at a mark, and values for average speed, distance traveled, gap to leader in seconds, velocity made good (VMG), estimated time of arrival at the next mark and current speed over ground. It was prototypical in many regards but regardless was considered an improvement for the commentary. The sailors liked it too because for the first time they could see numerical evidence of their choices of speed over distance.

After Kieler Woche 2011, the architecture changed. We removed the Python engine and used the Google Web Toolkit instead to render the Web UI directly in the Java process. A first new live leaderboard with this approach was shown at the IDM Travemünde 2011 and later at the MdM Hamburg 2011 events. Over time, the solution learned to manage multiple leaderboards, combining historic race analysis with live tracking. Particularly the accumulation of historic race data will require changes in the architecture in the near future to support this use case better.

A Google Map visualization, originally intended primarily for debugging purposes, matured to a useful tool used by commentators and spectators alike, combined with charts showing wind and competitor data, and of course the traditional live leaderboard. The leaderboard itself received various enhancements over time, including data about maneuvers such as tacks, jibes and penalty circles, and additional figures such as the average cross-track error which under shifty wind conditions in some boat classes may be an indicator for the risk taken by a competitor. Some of these figures turned out to be quite expensive to compute. Therefore, in a few cases we deviated from the original approach where everything was computed on the fly upon receiving a request. Instead, the more expensive calculations in live mode now happen asynchronously in the background, and client requests are fulfilled with whatever the most current result for these figures is.

The REST/JSON APIs offered by the Java back-end have been exploited by at least two additional show-case scenarios. Already in 2011, Business Objects Dashboards displayed data extracted through these interfaces in various analytical views. In 2012, the interfaces started to be used for repeated extraction of data into a HANA database on top of which Experience UI technology is now used for visualization with sophisticated analyses.

In 2012, a mobile application to support the race committees in their functions has been developed using largely the same architecture. Although the server for this app currently runs in a separate process, it uses largely the same code base, versioning repository and build process. We plan to integrate it with the SAP Sailing Analytics soon. A first loose coupling will allow users of the mobile app to send wind data entered on a mobile device into the SAP Sailing Analytics back-end where it augments the wind-based calculations. Later, we plan to integrate the mobile app even closer so that it supports race officials in laying and moving marks, changing the course layout as well as detecting and announcing disqualifications.

The remainder of this document explains the key architectural principles on which the SAP Sailing Analytics have been developed. It is to be considered a snapshot of the status quo, as documented by the time stamp in the document's header.

# Runtime Environment

## Linux, Java, OSGi/Equinox

By and large, the SAP Sailing Analytics are a web application implemented using Java technology. The application's components are OSGi bundles running in an Equinox OSGi container. Some bundles offer static web content or dynamic content in the form of servlets. Those bundles are implemented as OSGi Web Bundles which we consider a simple and elegant way to meet web standards using an OSGi-based implementation. A Web Bundle's main extension compared to a regular OSGi bundle is the presence of a web.xml descriptor in the WEB-INF top-level folder where servlets and static content can be declared and mapped to URLs.

Our current server deployment uses a 64bit Java7 Hotspot virtual machine and runs on a 64bit Linux CentOS distribution. We have a single host (sapsailing.com) which runs a number of Java VMs, some to offer the application in different development stages (dev, test, prod, ...), some to perform specific tasks such as replicating UDP wind data to the various server processes, or a process to store data received from the SwissTiming connector durably while forwarding that data to a server VM requesting it.

The various processes run in "screen" sessions to which, once connected to sapsailing.com with an ssh client, users can gain access using the "screen -rd" command. We consider replacing "screen" by "tmux" which is much more flexible and powerful and reported to be less buggy. This will allow us to launch the processes required in tmux sessions automatically after an operating system reboot.

For the OSGi containers by convention we have one directory under */home/trac/servers/* per deployable branch (*dev*, *test*, *prod1*, *prod2*). In those directories we have copies of the "install" script from the git's *java/target* folder. Running it after a successful product build on the branch corresponding to the current directory will copy the compiled product to the server directory. Running the *start* script will then launch the respective server instance. A safety check in the *install* script avoids accidentally overwriting a server directory with a non-matching product version by comparing the directory name with the branch name checked out under */home/trac/git*.

## Database

By and large, we use a database to recover from a server restart. Once started, most data managed by the application is kept in main memory. We currently use MongoDB as our database. Different DB instances belong to different server instances. This allows us to cleanly separate development and test data from production data.

The GPS tracking data is currently usually fetched from the tracking provider in case the provider stores it persistently. While this avoids redundancies and ensures up-to-date versions of the tracking data are used, it also creates a strong dependence upon the provider's system availability and may cause performance issues when many GPS tracks need to be reloaded after a server restart.

We therefore consider using our database also for replicated versions of at least the "archived" tracks where further changes on behalf of the tracking provider are unlikely.

## Google Web Toolkit (GWT)

The web UI is built using the Google Web Toolkit (GWT). This allows us to share code between UI and back-end and gives us the power of the regular Eclipse Java tools for code understanding, debugging and agile manipulation.

The GWT application communicates with the server using GWT RPC which, in the back-end, is implemented as a so-called RemoteServiceServlet which again is exposed by means of an OSGi web bundle. This servlet accesses the core application through an OSGi service (RacingEventService) which is hooked up in the OSGi service registry.

We try to keep important styling information separate in CSS resources which can be manipulated by web designers more conveniently than the Java source code. We balance this with the benefits of the Java sources' traceability which does not exist really for CSS resources where everything is just a string.

## Tracking and Wind Sensor Connectors

To receive GPS and wind data in near real time, some network programming becomes necessary. Depending on the technology and provider used, a combination of HTTP, TCP and UDP connections is required to obtain live data. Particularly the UDP connectivity was the reason why deploying our solution to SAP NetWeaver Cloud was and still is difficult.

We try to isolate connectors to a particular technology or tracking provider so that the core application doesn't depend on a particular provider. Among other things, this leads to an architecture in which separate bundles encapsulate the connectivity components for each provider. There are still a few minor pieces of code in the UI area where this separation hasn't been completed yet and where the UI component knows about the concrete GPS tracking providers currently supported. We have plans to change this such that simply by deploying a tracking provider's connectivity bundle the back-end picks it up and makes it known all the way into the front-end.

### TracTrac Connector

TracTrac offers a Java client to ease the connection to their back-end systems. This client is provided as a JAR file and is referred to as the "TracTrac Client Module" (TTCM). The client offers convenient access to both, historic ("stored") data and live data which is pushed to the client.

The TracTrac system serves JSON documents, one per event, which provides an overview of the races tracked by the system. This list is visualized in the TracTrac tab of the AdminConsole when the "List Races" button is pressed after having entered a correct JSON URL. The document also contains details about the connectivity parameters which we read and use by default.

TTCM requires access to specific TCP ports (usually germanmaster.traclive.dk:4400/4401). This is to be considered when configuring a firewall. It is generally possible to tunnel these connections through an SSH tunnel. TTCM is fairly resilient to network disruptions and keeps trying to re-connect. Once connected, the push service works pretty reliably during live events.

It is worth noting that the TracTrac architecture can lead to out-of-order delivery of messages. For example, if a tracker loses network connectivity for some time, it will send the GPS fixes recorded during the outage once it re-connects. Those fixes can affect past analyses such as the wind estimation. Also, mark rounding times can change over time. Whenever the TracTrac server computes an improved mark rounding time, it will push the update to all registered TTCM clients, possibly updating previous mark rounding time estimations. Since such updates have a major impact on many calculations, it was one of the key reasons we originally decided to store only the facts and calculate all derived figures from those facts on the fly.

We map the TracTrac domain concepts to our domain concepts in an adapter we call the *DomainFactory*. It keeps track of the mappings performed so as to not create duplicate domain objects in our application for the same TracTrac competitor, buoy, waypoint or race. These canonicalizing mappings are–together with the use of immutable master data objects–at the same time currently one of the annoyances in the architecture. When master data changes on TracTrac's side, our domain objects currently aren't properly updated because they are immutable, and replacing them would not be an easy task. See also <http://bugzilla.sapsailing.com/bugzilla/show_bug.cgi?id=596> for a more detailed discussion of this problem.

TTCM has a notion of "events" which can differ from what our domain model calls an "event." With TracTrac it is possible to track multiple sailing events and regattas within a single TracTrac event. However, in our domain model, a sailing event such as "Kieler Woche 2012" is a single event.

The connector runs a number of threads for each race tracked: one per type of message received. Those types are the race course definition together with the list of competitors, the raw competitor GPS fixes, the mark positions, start/finish times and the mark rounding times. In retrospect, this design is not ideal for a number of reasons. While it keeps each receiver for each type of information fairly clean, affording at least five threads per race is quite a lot and would not be needed if we handled the receiving of each message synchronously in the callback provided by TTCM.

A particular aspect in the connector's design to re-consider is the life cycle of the *RaceDefinition* objects. They are currently only created *after* the race course definition has been received. All callers required to access the *RaceDefinition* object are currently suspended until the course layout has been successfully received. If for some reasons this process times out

### SwissTiming Connector

Other than in the case of TracTrac, the SwissTiming *SailMaster* system broadcasts packets containing position and mark rounding information to a pre-determined set of hosts. There is no API to obtain old, stored data. Packets missed during live transmission are lost, except for a tricky, unconfirmed and yet untested process of receiving database dumps from SwissTiming at a later point.

Therefore, we have decided to implement a database buffer for the SwissTiming connector in such a way that a very simple, hence robust Java process is solely responsible for receiving, forwarding and durably storing the information packets broadcast by SwissTiming (see class *StoreAndForward*). The actual connector running in our back-end then connects to this process for the forwarded live packets while loading the packets already received so far from the persistent store. Based on packet numbers, the original sequence can be restored while live packets keep coming in.

It may be worth noting that SwissTiming SailMaster systems can operate in one of two modes. Either they offer inbound connections on an IP address and port. In this case, requests can also be sent to the SailMaster instance, e.g., to find out the set of races currently managed by that instance. Or the SailMaster system runs behind a multiplexer which collects messages from several SailMaster systems and broadcasts them to a previously determined set of IP addresses and ports. In this case, no requests can be sent to any of these SailMaster systems because the multiplexer couldn't identify the individual SailMaster instance to which to forward the request. For this reason, the connector distinguishes between these two modes and won't try to send requests if the SailMaster system runs behind a multiplexer.

The current implementation offers methods on the *SailMasterConnector* interface, allowing clients to send requests to a SailMaster instance capable of handling them. However, the connector doesn't currently use any of those, except for the *getRaces()* call which is used if available to determine the set of races managed by the SailMaster instance. Future versions should consider obtaining more information by explicit requests if possible.

### Expedition Connector

We like to receive wind data live from the race course(s). For this, we can install wind measurement devices on vessels such as the start vessel or RIBs following the field or dedicated solely to the task of wind measurement. Currently, we use devices that can be hooked up to a Panasonic Toughbook on which we run a software package called *Expedition* (see <http://www.tasmanbaynav.co.nz/>). This package is capable of receiving sensor data from a variety of different devices, including a Nexus on-board controller and various sensor devices connected directly to the laptop using, e.g., USB.

If sufficient information is available for Expedition to determine the sensor speed (either by a water log or by availability of GPS data), it can infer the true wind speed and bearing from the apparent wind speed/bearing measured.

Expedition can be configured to transmit one or more of the values it received or inferred across a network connection. The data is prefixed with a so-called "boat ID" which can be configured in the Expedition software. We use Expedition's capability to send the sensor data through UDP to port 2012 on the sapsailing.com host. We use a simple UDP mirroring process implemented in Java (see class *com.sap.sailing.expeditionconnector.UDPMirror*) to forward the messages received to the various server instances, each listening on a different UDP port.

Those UDP messages are received and analyzed by an active instance of class *UDPExpeditionReceiver*. Those that can be parsed successfully are used to record wind fixes in a tracked race with which the receiver is associated.

## Result Importers

For the operations of many regattas, software solutions are already in place. Those are used to manage competitor lists, fleet assignments, boat class to race course area assignments and of course scoring and ranking. Usually, they are the single source of truth also for the race committee and the sailors.

The SAP Sailing Analytics primarily base their leader boards on the tracking results. However, the tracking not always reflects the actual scoring as decided by the race committees. We therefore are interested in the ability to import the official results into our solution so that in addition to the GPS tracking data we have a copy of the truth when it comes to the scoring process. This is particularly exciting during the last race of a regatta when we can augment the official scores with the live tracking of the race currently ongoing.

We have provided an interface to import official results into our leader boards. Result importers implementing the *ScoreCorrectionProvider* interface can register with the OSGi service registry and are dynamically discovered. Such a provider can tell for which events and boat classes and dates it has "score corrections" which is a more technical term for "official results."

We currently have *ScoreCorrectionProvider* implementations for the 2011 and 2012 Kieler Woche result system (see bundle com.sap.sailing.kiworesultimport), for the french "FREG" system (see bundle com.sap.sailing.freg.resultimport) and for the Extreme Sailing Series (see bundle com.sap.sailing.ess40.resultimport). In their activators, these bundles register an instance of their score correction provider implementation with the OSGi service registry. This way, more importers can easily be added, potentially even at runtime, simply by deploying and starting another result importer bundle.

### Kieler Woche Result Importer

There is no online access to the Kieler Woche regatta system. However, we were able to agree an FTP result export with the system provider, b+m. The export format is a ZIP file containing numerous XML and PDF files. The XML files are then analyzed and provide the contents for the score corrections. We created a dedicated account for the FTP export on sapsailing.com ("kiwo"). The FTP export ends up in /home/kiwo, and all server instance directories under /home/trac/servers/(dev|test|prod1|prod2) contain a link "kiworesults" pointing to /home/kiwo.

The score correction provider implementation scans the "./kiworesults" directory for the ZIP files and offers their contents through its API.

### FREG Result Importer

At the 505 Worlds in La Rochelle we were faced with an HTML export format. Unfortunately, there was no stable URL scheme from which to obtain the HTML documents exported from the regatta management system. Therefore, we decided to leave the URL configuration to the administrator of our solution and added a tab "FREG URLs" to the AdminConsole. The documents reachable through the URLs added to this page will be scanned by the FREG score correction provider.

### ESS40 Result Importer

The Extreme Sailing Series currently manages their results through sailracer.org. The series has their own iPad app to capture the finish line passings. This app produces a CSV file managed on the series' web server before it is converted and uploaded to the sailracer.org server. We get access to the CSV files and a document listing the CSV files available for the series. Those then feed into the score correction provider.

# Basic Architectural Principles

## Domain Model

We have created a set of interfaces that represent the concepts of the domain of sailing races. Typical abstractions captured by these interfaces are, e.g., *Regatta*, *Series*, *Course*, *Leg*, Waypoint, *Buoy* and *Competitor*. There are also interfaces describing more general concepts not necessarily specific to sailing, such as *GPSTrack*, *Position* and *Distance*.

Instances of classes implementing these domain interfaces are generally created using a *DomainFactory* instance. However, in some special cases constructors of domain classes implementing the domain interfaces may also be invoked directly.

We have tried to split the domain interfaces into such describing "master data" and others describing "transactional data." The master data interfaces can be found largely in the *com.sap.sailing.domain* and *com.sap.sailing.domain.common* packages. "Transactional" data is largely obtained through tracking devices which is why most of those interfaces are found in the *com.sap.sailing.domain.tracking* package. For some of the master data domain interfaces, we have corresponding "Tracked..." counterparts that capture tracking data that pertains to the respective master data object. For example, *TrackedRace* describes the tracking data for a race. It has *TrackedLeg* as one of its constituents, corresponding to the *Leg* objects contained by a race's *Course* object.

Where possible we have tried to keep objects immutable. This has several advantages, particularly when caching such objects and when replicating and propagating and displaying in a user interface. However, this rigid approach also leads to problems, particularly if the underlying assumption that an object doesn't change turns out not to be true in all cases. For example, we have modeled the *Competitor* interface as immutable. However, if the competitor data was obtained through a tracking service provider who made a mistake during capturing the competitor data which is later corrected, having the *Competitor* as immutable is a real problem.

For some of those concepts that allow for changes to be applied, we have distinguished between the reading interface and the modifying interface. For example, *DynamicTrackedRace* is the modifiable counterpart to *TrackedRace* which offers only reading operations.

The domain model is independent of tracking service provider specifics. When the system receives data from a tracking provider, the specific messages are converted into domain objects. To avoid duplicate domain objects for the same entity (e.g., a *Competitor*), the connectors (see also ) typically keep mappings that allow the connector to retrieve and use an existing domain object instead of creating new ones. The challenge with these mappings is to avoid memory leaks. Currently, the existing mappings only maintain domain objects that have a small memory footprint. We need a good strategy for releasing objects from these mappings. Ideally, the life cycle of the objects in such caches should be coupled to the life cycle of other domain objects. For example, a competitor object may be released if all races and regattas and leaderboards possibly using this competitor have been garbage collected.

## In-Memory Architecture and the Database

The application generally deals with two sorts of data. One is the sensor data originally created by a set of physical sensors such as GPS trackers, wind measurement devices or the sensors included in smart phones and other mobile devices. The other is master data and meta-data captured and maintained by administrators and users, such as the leaderboard configuration data, connectivity data for the tracking providers, or official scoring results imported from external sources.

For the sensor data, the time at which the data is received is never the same as the time at which the sensor data was valid. Sensors don't predict the future but measure some present value. The transmission from the sensor to the server adds a rather unpredictable latency. In the worst case, the sensor's transmission unit fails, and the data can only be imported into the server once the sensor is back on shore. With this in mind, the server's view of reality is partial and lagging, and history may be re-written at any point in time if a sensor decides to deliver its data later than most other sensors. There is no precise synchronization across the set of sensors used at an event. At best, there is a pre-configured maximum delay at which trackers make an effort to deliver their data. However, this may also fail, for example, if trackers lack network connectivity when they would actually be due to send their data. Tracking providers may also decide to re-compute some derived data which our server receives. For example, some tracking providers send data about when they think which boat passed which mark. The provider may change this at any time, sending an updated list of mark passing times. This can, e.g., happen if the course layout was changed on short notice, and the course update didn't make it into the server in time. Once the course layout change is then updated to the system, the mark passing times will be re-evaluated, and updates to the previous mark passing times will result.

These circumstances suggest an architecture which basically records the sensor facts and dynamically aggregates all derived information on the fly. This is how we started. The more complex the rules for deriving interesting figures from the sensor data grew, the more computational resources the on-the-fly aggregation required. In particular, three algorithms turned out to be quite expensive to carry out: maneuver analysis based on the recursive Douglas-Peucker algorithm; wind estimation based on the boat tracks, assuming that boats on different tacks use roughly the same beat angle to the wind; and the average cross-track error which computes a projection of each boat's position to the wind direction. In addition to those, aggregations of distances traveled incur lots of duplicate work if done naïvely. Usually, GPS fixes are appended to the end of a track. Therefore, the distance traveled would re-compute the distances traveled from the start to the last but one fix which is obviously unnecessary.

The role of the database in all of this is currently largely for recovery purposes only. It stores the administrative information such as which leader boards have been defined, the structure of regattas, explicit score corrections, and the association of tracked races with leaderboard columns, to name a few. Additionally, it stores the wind measurements because those would be hard to recover when the server needs a re-start. For the GPS tracking data, however, we currently try to recover them from the tracking provider where possible.

While this works well for the TracTrac connector, in case of the SwissTiming connector this is not possible at all. Therefore, we developed an additional component (see also *SwissTiming Connector*) that can record the SwissTiming messages received in the database so they can be retrieved from there after a server restart.

One key consequence of this architecture is that if two server instances share a common database then usually the second instance doesn't see the first instances writes to the database before the second instance is restarted and recovers from the joint database[[1]](#footnote-1).

## Caching where Necessary

Based on the performance implications explained above, we decided to carefully introduce caching where it was absolutely necessary. We added a cache for the results of the maneuver analysis (see *TrackedRaceImpl.maneuverCache*). Another cache exists for the wind estimations derived from the boats' GPS tracks (see *TrackBasedEstimationWindTrackImpl.cache*). Yet another cache was introduced for the average cross track error (XTE, see *TrackedRaceImpl.crossTrackErrorCache*) and for the distances traveled (see *GPSFixTrackImpl.distanceCache*) as well as for the maximum speeds (see *GPSFixTrackImpl.maxSpeedCache*). Another trivial cache in *TrackedRaceImpl.directionFromStartToNextMarkCache* is used to speed up the query for the direction of the first leg which is frequently used in races starting upwind.

While those caches speed up requests on *TrackedRace*, even larger performance gains can be achieved by caching the higher-level aggregates requested by many clients. In particular, many clients request the same leader board in live mode. Then, having the server determine what the "live" time is instead of using the client's clock, and just returning the latest contents for the leader board requested from a cache has increased the application's scalability significantly. The leaderboard caching is implemented in the classes *LeaderboardDTOCache* and *LiveLeaderboardUpdater*.

## Implementation Patterns for Caches

A cache is like a materialized view in a database. The greatest challenge with a cache is to keep it consistent. The challenge becomes greater if concurrency is added to the mix, and tens or even hundreds of sensor fixes update the data structures that provide the input for the calculations filling the caches.

In the SAP Sailing Analytics, when introducing a cache we usually need to consider the following questions:

* When do which parts of the cache need to be invalidated?
* Should invalidated cache contents be re-calculated straight upon invalidation, or should this wait until the next relevant request is received?
* Can clients tolerate stale cache contents, permitting to update the cache contents in a background thread?

The invalidation logic depends directly on the computation rules for the value being cached. The cache needs to act as observer of those values so that it receives a notification in case a value changes that the cache content depends on. A useful observer pattern is offered by *TrackedRace* which offers the registration of *RaceChangeListener* objects as observers. There are many more such patterns in place, e.g., the *RaceColumnListener*, *CourseListener*, *RaceListener* and *RegattaListener*.

Whether or not to re-calculate cache contents right after their invalidation usually depends on the access patterns and frequencies compared to the cost and duration of re-calculating. Especially in live mode, we pro-actively keep the *LiveLeaderboardUpdater* running as long as there are requests for live leaderboards received. Using a timeout (see *LiveLeaderboardUpdater.UPDATE\_TIMEOUT\_IN\_MILLIS*) the updater stops re-calculating cache contents if for that time no request has been received. Other cache contents, such as for the cache for the direction from the start to the next mark, are only re-calculated when the next request is received because a single re-calculation doesn't take long, compared to the number of times it can then be used from the cache and compared to the overall round-trip time of the request to which this single value contributes.

For some values a client may be able to tolerate a certain lag relative to the latest data received. This also has to be evaluated against the background of the delay in the sensor data transmission, starting at the time point when the sensor picks up a value until the value has arrived in the SAP Sailing Analytics. If this delay is significantly longer than the time required to re-calculate a cache entry based on a new sensor value received, then it is not all that bad to keep pretending for a while that the last cached value is still the best we have until a short time later a cache update is performed, taking into account the new value.

If a cache can be used by clients not worrying too much about slight staleness of data, cache updates can be performed in background threads, and cache contents that we know already are outdated can be continued to be served until the re-calculation has completed.

The *SmartFutureCache* class serves as a base for implementing caches that can be updated by a background task. Values in those caches are only updated, never removed. Clients can choose whether they want to wait for the latest ongoing calculation or if they just want to get the last value available (which may be a bit stale).

## Approaches to Locking: "synchronized" vs. ReentrantReadWriteLock

User requests, receiving sensor events and updating caches all happen concurrently, using many threads. Additionally, some user requests are split into many tasks, each of which can be assigned to a separate thread. It is instrumental that no inconsistencies occur due to this high level of concurrency. The usual approaches to locking have to be considered; only that the architecture of the SAP Sailing Analytics is not primarily one using a database with support for transactions, but an in-memory architecture with competing reads and updates.

We started out by using the Java built-in object monitors ("*synchronized*") to protect code regions from concurrent access of readers and writers or multiple readers. Being a basic language construct, the tool support for *synchronized* is great. The Eclipse debugger can visualize deadlocks right away, once they occur (and they did of course occur).

There are two well-known problems with using *synchronized* as an approach to locking:

* Locking is bound to the lexical code structure.   
  Code regions using *synchronized* have to be specified as a block of Java code. While this ensures that locks are properly released once the execution leaves this block of code, it makes hand-over-hand locking impossible, and it makes locking a variable set of objects before executing some block of code impossible.
* It is not possible to distinguish between readers and writers so that multiple readers could read concurrently.   
  This fact reduces the amount of concurrency possible to only one concurrent reader per lock object. For some situations this is just not sufficient.

To increase the amount of concurrency across multiple readers, we decided to use *ReentrantReadWriteLock* instead of *synchronized* where we had identified bottlenecks. This helped to increase concurrency a lot. However, we had to sacrifice two things that were so convenient with *sycnhronized*, namely the tool support (tools don't understand deadlocks between readers and writers which hold technically distinct primitive locks) and the implicit correctness of releasing the lock again. The latter boils down to establishing a discipline of always releasing each lock obtained in a *finally* clause. Although no longer enforced by the lexical structure, it isn't too difficult to execute this discipline.

The lack of tool support, particularly for deadlock detection, hurts worse. Some deadlocks that occurred with the use of *ReentrantReadWriteLock* were obvious and easy to find. Others, however, were very hard to find, particularly if they happened in a Java VM that was not running in debug mode and hence didn't allow attaching an external tool that would have helped in identifying the various call stacks and lock ownership relationships.

To alleviate this problem, we at least introduced excessive tracing in case a lock cannot be obtained for some period of time. The trace output contains all reader and the writer thread (if any) including their full stack traces. This has already helped identify one further deadlock which occurred since the introduction of the traces.

Generally, one of the most important locks in the system is that of a *Course* object. When a course changes, e.g., by the addition or removal of waypoints, many dependent data structures will need to be updated. Therefore, in turn, operating on those data structures requires the caller to hold the *Course*'s read lock. This will hold back a course update until no reader is working with the course or any derivatives.

## Scale-Out through Replication

Scalability becomes an issue as the number of concurrent users increases and the requirements regarding availability grow. We think that replicating a server instance so that the replica can respond to client requests will help towards both goals: handle more concurrent users and increase availability.

### Master/Replica Distinction

There are various ways to implement server replication. Among the easiest is a landscape that distinguishes a "master" from a "replica" side. In such a set-up, changes are injected into the master first, from where they are replicated to all replicas. No other changes are permitted on the replicas in this easy set-up, leading to eventual consistency once the flow of events on the master stops.

### Operational Transformation

With a distinction between master and replica it also becomes possible to apply a technique called *Operational Transformation* (OT). It is an approach leading to eventual consistency between one master and multiple replicas, supporting changes injected on *both*, master *and* replicas. This may become an interesting option when discussing a dying master and the replicas negotiating a new master.

The code base already contains an OT implementation (see package *com.sap.sailing.server.operationaltransformation*). The operations used for replication are prepared to interact with this framework, but the implementation of the transformation rules are largely not yet implemented except for a few tests in the area of leaderboard-related operations.

### Implementation of Operations, Services, and Events

The operations used for replicating changes are so far all located in the package *com.sap.sailing.server.operationaltransformation*. There are three ways in which these operations are being used, two of which need urgent consolidation.

#### GWT Service Constructing and Applying an Operation

In some cases, an instance of the class *SailingServiceImpl* which handles incoming GWT servlet requests constructs an operation object and applies it to the *RacingEventService* directly by calling *getService().apply(operation)*. This way of using operations should probably better be encapsulated and hidden behind the *RacingEventService* interfaces, as we have done for many operations already, as described in the following section.

#### RacingEventService Constructing and Replicating an Operation Triggered by a Client Request

Several *RacingEventService* methods that are relevant for replication do their job and afterwards produce an operation object submitted to the replicator. The receiving replica applies the operation to its own *RacingEventService*. Most of these operations are implemented such that they call the same method that produced the operation on the master. While this again constructs an operation on the replica and submits it to the replica's own replicator, no transitive replication happens usually because a replica has no further replicas.

It would probably be better to consolidate the two ways in which explicit requests use the replication architecture. It would be nice if the same operation implementation was used on the master and the replica side to carry out the actual change on the *RacingEventService* instance. Ideally, the *RacingEventService* would offer dedicated methods, not for external use but for use by the operation implementations. Those dedicated methods then wouldn't have to worry about replication. Methods exposed by *RacingEventService* should then guarantee replication by producing the operation locally, applying it (which uses the dedicated, non-exposed methods) and replicating it. The receiving replica would then again just apply the operation, with no further replication being implicitly triggered.

#### RacingEventService Constructing and Applying an Operation Triggered by an Event

The system receives event notifications that lead to a state change requiring replication. For example, when a GPS fix arrives from a boat tracker, the fix needs to be recorded locally and has to be propagated to all replicas. This currently happens by letting the *RacingEventService* observe the data structures that change due to such events, for example, the GPS and wind tracks. The *RacingEventService* maintains a *RaceChangeListener* for each tracked race. Whenever a change worth replicating occurs on any of these races, the *RacingEventService* forms an operation and passes it to the replicator.

When thinking about the necessary consolidation of the different approaches to replication, we may even want to go as far as having these types of low-level, fine-granular changes be funneled through the *RacingEventService* interface in a consistent way, such that the service doesn't need to listen to low-level data structures for replication purposes anymore but instead can use the explicit calls to exposed methods to assemble an operation which, when applied locally, updates the respective data structures and can be replicated consistently.

### Replication and the Database

So far we have tested the replication only with replicas that maintain their own database. This database, however, is not read from after the node has been defined to be a replica. When becoming a replica, a *RacingEventService* clears all its internal state and requests an initial load of data from its assigned master node.

When operations arriving at the replica are executed, many of them call features of the *RacingEventService* which also update the replica's local database. So far, we have not carried out any tests regarding the completeness or consistency of the resulting database image. Keep in mind that the replica may have started out as a system with its own particular database content. When receiving the initial load, this database content is not removed. Probably, a partial update and new inserts may result. Note also that the initial load does not at all update the replica's database. It is probably safe to assume that a replica's database will be in a questionable state that the replica should no longer rely on. It may serve as an accidental back-up in case the leaderboard score corrections have been lost on the master and happened to be replicated through an operation to the replica so that they updated the database there. But this is nothing planned or anything to rely on.

A larger discussion about this topic has to be started if we start to roll out replication into our production systems at a larger scale.

### Open Issues

Besides the database issues on the replica which were discussed in the previous section, two more urgent issues exist with the current form of replication.

#### Dead Master

When a master with one or more active replicas dies (gets stopped, crashes, ...), the replicas will still be able to handle client requests. However, the replicas won't receive further updates from the master anymore.

When the master is started again, we currently have to load the races into the master's main memory again. This currently works by means of the regular connectors to the tracking providers, as if a live race were being tracked. We currently don't have a persistent local copy of the tracking data received from the tracking providers.

When a master starts, it has forgotten about its replicas (which is something we may consider to change in the future). Therefore, the replicator is not configured to listen to operations emitted by the *RacingEventService* yet. Only once a replica is explicitly added again to the master will the master start to emit the operations again to the message queuing system which then distributes them to the replicas again.

Therefore, when loading all tracking data again into the master, it depends on whether a replica was registered with the master again before starting the loading process. If not, the tracking data recorded between the master's death and the time when one replica is registered is lost for the replicas unless they choose to do a full initial load again.

Solution approaches may include a persistent representation of the replication topology as well as a heartbeat check performed by the replicas to see if the master is still alive, including some sort of probing for the master's recovery, followed by an automatic re-registration for the operation stream.

#### Dead Client

If a replica dies, the master won't know. This is no problem as long as there are still replicas surviving. But if the last replica has died, the master will not stop pushing the operations into the message queuing system. The master loses track of which replicas are still actively registered.

We haven't yet measured the overhead of pushing the operations into the message queuing system, but it seems good to avoid it if it's not needed.

Therefore, a heartbeat mechanism performed by the clients to check for the master's liveness can also be used by the master to see which clients are still alive.

## Preference for Immutable Value Objects

It is a general pattern in the application to try to keep objects immutable where it reasonably makes sense. Offering a setter for a field is a huge and often underestimated commitment. It allows clients to modify the object more or less at all times if no locking precautions are taken. Modifying an object also needs to be aligned with the replication concern, making sure that if the modification is relevant for the replicas, an operation needs to be implemented that carries this change on to the replicas consistently.

Object immutability has other advantages as well. The application is highly concurrent, using many concurrent threads. Setters always need to ensure that the object is still in a consistent state when the setter returns. If multiple setter calls are necessary to bring the object into a consistent state again, this needs to be hidden behind a common operation that carries out all necessary changes and takes the locking steps necessary to avoid concurrency issues.

A typical pitfall with mutable objects is to assume that an object is not visible to other threads when starting to modify it. Guaranteeing this is very hard. Mostly, it's not even the case, so other threads may be able to see an object that is under modification by one thread. In such a situation it is impossible for one thread to rely on its changes to prevail because other threads may perform competing changes using the setter on this object at all times.

Therefore, if it is by any means possible, making all fields final and passing all values necessary to the constructor is to be preferred over offering setters. This lets the object pass around freely without the risk of race conditions and transactional inconsistencies.

# Development Environment

## Git and Our Branches

Our main Git repository lives at *ssh://<user>@sapsailing.com/home/trac/git*. For those working in the SAP corporate network and therefore unable to access the external sapsailing.com server using SSH, the repository contents are replicated on an hourly basis into *ssh://dxxxxxx@git.wdf.sap.corp:29418/SAPSail/sapsailingcapture.git* where *dxxxxxx* obviously is to be replaced by your D- or I- or C-user. You need to have an account at <https://git.wdf.sap.corp:8080/> to be able to access this Git repository.

Small, obvious and non-disruptive developments are usually carried out immediately on our *master* branch. This branch is configured such that Maven can be used to run the tests inside the SAP corporate network. The *master* branch is never deployed onto the sapsailing.com server and hence has no corresponding */home/trac/servers/* subdirectory.

If a change looks reasonably good on the *master* branch and related JUnit tests or manual UI tests have succeeded locally, it is permissible to merge the *master* branch into the *dev* branch and run a central build on sapsailing.com. The *dev* branch is configured to run the Maven tests with direct Internet access. It can therefore also be used to run the tests locally if connected to the public Internet.

Ideally, the build should be run including the test cases. If the tests succeed[[2]](#footnote-2), the branch can be installed and the corresponding server instance can be restarted. The branch can then also be promoted to the next level (*dev*-->*test*-->*prod1*/*prod2*). Note, that currently re-starting a server instance may require re-loading the races that were previously loaded, particularly for the *prod1* and *prod2* instances, because several externally-announced URLs point to them.

We typically promote the changes to the next branch by also merging the current *master* branch into the next-"higher" branch. This should lead to equivalent results compared to merging the "previous" branch (e.g., *dev*) into the "next" branch (e.g., *test*). The branches differ largely in the configurations used for the servers, particularly the port assignments for the Jetty web server, the UDP ports used for listening for Expedition wind messages, and the queue names used for the replication based on RabbitMQ (see also ).

## Eclipse Setup, Required Plug-Ins

We use Eclipse as our development environment. Using a different IDE would make it hard to re-use the project configurations (*.project* and *.classpath* files which are specific to Eclipse) as well as the GWT plug-in which assists in locally compiling and refactoring the GWT code, in particular the RPC code.

The recommended and tested Eclipse version is currently Indigo (3.7). Colleagues have reported that they succeeded with a Juno (3.8/4.x) set-up as well.

To get started, install Eclipse with at least PDE, Git, GWT (use <http://dl.google.com/eclipse/plugin/3.7> as update site) and JSP editing support enabled. Eclipse Maven support is not recommended as in many cases it has caused trouble with the local Eclipse build.

## Target Platform

After the Eclipse installation and importing all projects under *java/* from Git, it is required to set the target platform in Eclipse (*Window* --> *Preferences* --> *Plugin Development* --> *Target Platform*). The project *com.sap.sailing.targetplatform* contains "Race Analysis Target (IDE)" as target platform definition. It uses a number of p2 update sites and defines the OSGi bundles that constitute the target platform for the application. If this is not set in Eclipse, the local build environment assumes that the developer wants to implement Eclipse plug-ins and offers the entire set of Eclipse plug-ins and only those as the target platform which doesn't make any sense for our application.

Major parts of our target platform are hosted on sapsailing.com as a p2 repository which makes it possible to have only one central target platform configuration used by everyone. The target platform can be re-built, e.g., after adding another bundle to it, using the script in *com.sap.sailing.targetplatform/scripts*. It then needs to be installed again to the */home/trac/p2-repositories* directory from where it is exposed as <http://www.sapsailing.com/p2> by the Apache web server. After such a change, all developers need to reload the target platform into their Eclipse environment.

## Maven Build and Tests

We use Maven to build our software and run our JUnit tests. The global *setting.xml* file to install in everyone's *~/.m2* directory is checked into the top-level Git folder. The checked-in copy assumes the developer is using Maven inside the SAP corporate network. If not, uncomment the *<proxy>* tag in the *settings.xml* file. See also section *Git and Our Branches* for details on which branch is configured to work in which network setup.

We have a top-level Maven *pom.xml* configuration file in the root folder of our Git workspace. It delegates to the *pom.xml* file in the *java/* folder where all the bundle projects are defined. We rely on the Tycho Maven plug-in to build our OSGi bundles, also known as the "manifest-first approach." The key idea is to mainly develop using Eclipse means, including its OSGi manifest editing capabilities, and keep the Maven infrastructure as simple as possible, deriving component dependencies from the OSGi manifests. See the various *pom.xml* files in the projects to see the project-specific settings. By and large, a *pom.xml* file for a bundle needs to have the bundle name and version defined (we currently have most bundles at version 1.0.0.qualifier in the manifest or 1.0.0.SNAPSHOT in Maven), and whether the bundle is a test or non-test bundle, expressed as the *packaging* type which here can be one of *eclipse-plugin* or *ecplise-test-plugin*.

Test plugins automatically have their tests executed during a Maven build unless the command-line option *-Dmaven.test.skip=true* argument is specified. It is generally a good idea to launch the Maven command using the *-fae* option which asks Maven to continue until the end, even if errors or failures occurred on the way, failing at the end if any failures occurred. This can save numerous round trips and is useful in case of known and temporarily acceptable test failures.

The Maven plug-in for the GWT compilation doesn't reliably perform a dependency check. It is therefore recommended to remove all contents of the *java/com.sap.sailing.gwt.ui/com.sap.sailing.\** folders (basically, all GWT compiler output) before launching the Maven build. A good command line for the Maven build from the *java/* subdirectory is this:

*rm -rf com.sap.sailing.gwt.ui/com.sap.sailing.\*; mvn -fae clean install 2>&1 | tee log*

It also creates a *log* file with all error messages, just in case the screen buffer is not sufficient to hold all scrolling error messages.

# Typical Development Scenarios

## Adding a Column to the Leaderboard

It is a typical request to have a new column with a new key figure added to the leaderboard structure. A number of things need to be considered to implement this.

### Extending LeaderboardDTO Contents

The content to be displayed by the *LeaderboardPanel* user interface component is transmitted from the server to the client using a *LeaderboardDTO* object. The class describes overall measures and data for the entire leaderboard, has overviews for each race as well as all detail figures for all legs. For example, the *LeaderboardDTO* object contains *LeaderboardRowDTO* objects, one for each competitor. Each such row object, in turn, contains *LeaderboardEntryDTO*s, one for each race column in the leaderboard. Those contain aggregates for the respective race/competitor as well as a list of *LegEntryDTO*s describing key figures for the competitors performance in each leg of the race.

Depending on the level (overall, race, leg) at which to add a column, a corresponding field may need to be added to one of *LeaderboardDTO*, *LeaderboardRowDTO*, *LeaderboardEntryDTO* or *LegEntryDTO*. It may, however, at times also be possible to derive a new value from other values already fully contained in the *LeaderboardDTO* object. In such a case, no change is required to the *LeaderboardDTO* type hierarchy.

### Filling LeaderboardDTO Extensions

The *LeaderboardDTO* objects are constructed in *SailingServiceImpl.computeLeaderboardByName* and its outbound call hierarchy. Looking at the existing code, it should become clear how to extend this. Usually, if really new measures are to be introduced, a corresponding extension to *TrackedRace* and its key implementation class *TrackedRaceImpl* becomes necessary where the new measures are computed, based on the raw tracking data and the various domain APIs available there.

When a new measure is introduced such that *computeLeaderboardByName* needs to access data which may change over time, it is important to make sure that the leaderboard caches in *LiveLeaderboardUpdater and LeaderboardDTOCache* are invalidated accordingly if the data used by the computations introduced changes. For example, if the new figure depends on the structure of the leaderboard, the cache needs to be invalidated when the leaderboard's column layout changes. Fortunately, in this case, such an observer pattern already exists for the leaderboard caches and can be used to understand how such a pattern needs to be implemented.

### Adding the Column Type

The *LeaderboardPanel* and its dependent components are the user interface components used to display the leaderboard. The underlying GWT component used to render the leaderboard is a *CellTable* with *Column* implementations for the various different column types. In a little "micro-framework" we support expandable columns (those with a "+" button in the header allowing the user to expand more details for that column), columns with CSS styles that travel with the column as the column moves to the right or the left in the table, as well as sortable columns. A special column base class exists to represent *Double* values. This column type displays a colored bar, symbolizing the relative value's magnitude, compared to the other values in the column.

A column class is implemented as a subclass of *SortableColumn*. If the column is to display a numeric value where comparing between values makes sense, consider using a *FormattedDoubleLegDetailColumn* which displays a bar in the value's background, indicating the value's magnitude. To make this column type widely applicable, its constructor accepts a *LegDetailField* value which is responsible for computing or extracting the value to be displayed from a *LeaderboardRowDTO* object.

A column which may itself have details should be implemented as a subclass of *ExpandableSortableColumn*, implementing the *getDetailColumnMap* method to specify the detail columns, as explained later in .

### Extending DetailType

The enumeration type *DetailType* has a literal for each column type available. The literal describes the column's precision as a number of decimals (not used for non-numeric column types) and the default sorting order.

To make the new *DetailType* literal displayable, *DetailTypeFormatter.format* needs to be extended by a corresponding *case* clause so that it supports the new literal.

### Adding to List of Available Columns

Mostly for the presentation in the leaderboard settings panel, the lists of available column types are maintained in class-level methods on *LeaderboardPanel* (*getAvailableRaceDetailColumnTypes* and *getAvailableOverallDetailColumnTypes*) *ManeuverCountRaceColumn* (*getAvailableManeuverDetailColumnTypes*) and *LegColumn* (*getAvailableLegDetailColumnTypes*). When adding a detail column to a parent column, the corresponding *getAvailable...ColumnTypes* method needs to be extended by returning the respective *DetailsType* literal so the column is offered in the settings panel.

### Adding to Parent Column's Detail Column Map

If the column added is supposed to be a detail of a parent column, the new column type needs to be returned by the parent column's *getDetailColumnMap* method. Check the *LegColumn.getDetailColumnMap* for details.

### Discussion

The current approach to extending the leaderboard panel by another column is unnecessarily laborious and contains a number of redundancies. In particular, adding the column to both, the detail column map and the list of "available columns" only used by the settings seems highly redundant. When constructing a *FormattedDoubleLegDetailColumn, the DetailType* literal is used three times: once for the key of the detail column map, then for the precision and default sorting order. This should be simplified.

Generally, there are too many special cases necessitating specific handling. Instead, it would be much better to have a homogeneous hierarchy of column types which automatically leads to the necessary results in *getDetailColumnMap* and the settings dialogs.

## Adding a *ScoreCorrectionProvider*

External regatta management systems can usually provide the official scores through some electronic interface. These interfaces come with a pull or push transport protocol which may range from direct TCP, HTTP to an FTP file transfer, and a content format which can be anything from a simple CSV format to a complex XML document structure. Using such interfaces results in the capability of importing the official scores as score corrections into our leader boards, thus aligning the tracking results and the official results.

As regatta management systems vary vastly and from event to event, we use an open and flexible architecture for integrating with them. Key entry point is the *ScoreCorrectionProvider* interface that needs to be implemented for an integration with a regatta management system. An instance of the resulting type then has to be added to the OSGi service registry. Usually, this happens in an OSGi bundle activator of a bundle dedicated to the result import from the regatta management system to integrate. Example:

Activator.*context* = bundleContext;

**final** ScoreCorrectionProviderImpl service = **new** ScoreCorrectionProviderImpl();

*context*.registerService(ScoreCorrectionProvider.**class**, service, /\* properties \*/**null**);

When the bundle has been activated successfully in the OSGi container, then when a user triggers a result import, the importer will be asked for its results available so that the user can select them.

See the Javadocs and the existing implementations of the *ScoreCorrectionProvider* interface for more details.

## Adding a Maintainable Property on a Leaderboard

### Domain Model

### Cache Invalidation

### User Interface

### Persistence Layer

### Replication

# Other Information Sources

1. Exception: the SwissTiming message store is queried by the *StoreAndForward* component when the connector asks for race data. But then again, the *StoreAndForward* component usually runs in a separate VM. [↑](#footnote-ref-1)
2. We currently see intermittent failures in the replication and swisstimingadapter test suites. It's always worth a look, but if those are the only tests failing, it is still acceptable to promote the branch to the next level. We're working on stabilizing these two test suites to ensure they always succeed if the software is correct. [↑](#footnote-ref-2)