Technology Overview

SPARKL®
Sequencing Engine

Dr Andrew Farrell
Software & Research Engineer

Introduction to the SPARKL Sequencing Engine

The SPARKL® Sequencing Engine is powerful technology for managing the behaviour of distributed systems, bringing transparency into the operations of enterprise and industrial systems. It brings reason to the provisioning of infrastructure used for implementing services, capturing application-level and infrastructure-level event logs in a uniform manner.

This enables activities such as **anomaly detection** and **root-cause analysis**, using clean event data across all layers of the service stack.

Reasoned Provisioning

SPARKL determines the actions that should be performed on service end-points (including creation and tear-down of the service artefacts themselves), and thus what data should flow between end-points.

A user can configure SPARKL in application terms through the definition of services, along with various operations that may be performed on those services. Whenever a transaction is initiated, SPARKL will plan its satisfaction using the specified application configurations. It will route messages between services according to these plans.

It tracks the transaction-based provenance of operations on end-points as well as the data that flows between them, as part of its **reasoned logging** approach. This is essential for requirements in regulatory compliance in banking, for example.

Microservices

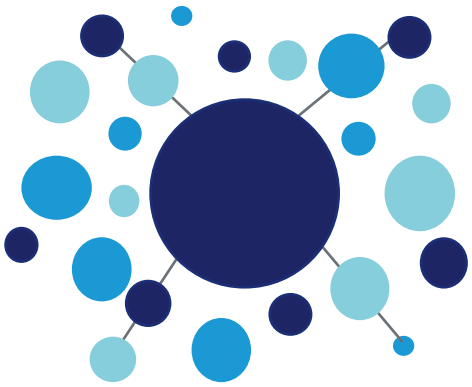
The use of microservices in the implementation of systems is increasing. Companies such as Netflix and Paypal were using older, more monolithic architectures, which made it difficult for them to add new functionality to their systems, and thus decided to adopt a more loosely-coupled approach.

Software developer and blogger Martin Fowler describes microservices as:

"...An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms.

These services are built around business capabilities and independently deployable by fully-automated deployment machinery. There is a bare minimum of centralised management of these services, which may be written in different programming languages, and use different data storage technologies."

Microservices should be as small as possible while still maintaining high cohesion overall - that is, related logic is placed within the same microservice, whilst unrelated logic should not be. This results in a typical layout according to business function at a fine level of granularity.



The use of microservices allows developers to better align the system architecture to the organization of the enterprise using the system. Each business function can be treated in a customised way - the system may implement different quality of service practices for the various functions, implemented as microservices.

The advantages of microservices are many: increased system resilience, fine-grained scaling, function replaceability and flexible composability, to name a few.

An ideal application of SPARKL is in the orchestration of microservices. The Sequencing Engine can reason when a microservice is needed at all - e.g. in the satisfaction of a transaction, or whether the number of instances of a microservice should be scaled out because of long transaction times.

Microservice Orchestration with Data Provenance

Service orchestration in SPARKL is primarily about supporting application flows. This means that, in the limit, service infrastructure may be provisioned and torn-down per individual call made against SPARKL, as depicted in **Figure 1**. In reality, there will be a balance between per-operation service artefacts, those that last for the lifetime of the overall service deployment, and others in between.

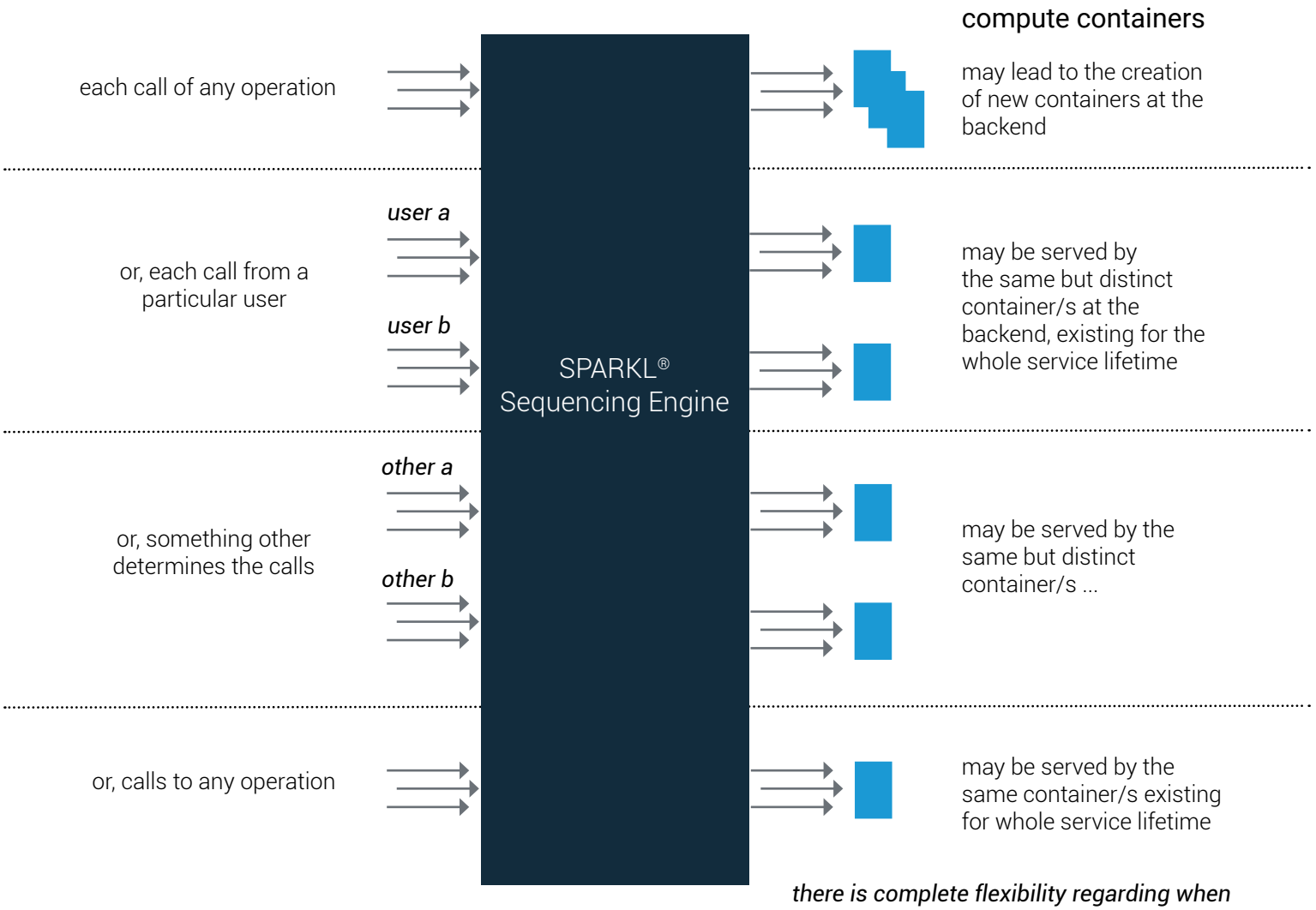


Figure 1 - Application-flow based Service Orchestration in SPARKL

Provisioning per application-flow may make sense for:

- Individual transactions that are long-running (thus reducing infrastructure set-up latency and costs)
- Per-user or some other determinant, in order to fine-tune performance
- Contexts that are sensitive in terms of its security requirements, for which operating in the context of **clean** infrastructure would be advantageous.

Application-flow provisioning works particularly well in the microservices model because of the fine-grained nature of the services themselves. SPARKL is able to reason when a microservice is needed at all - e.g. in the satisfaction of a transaction, whether the number of instances of a microservice should be scaled out because of long transaction times, or to implement different qualities of service for different users at different times.

A transaction can be executed on a system using SPARKL, according to these two patterns:

- The `solicit->response` pattern describes the solicitation of a transaction to achieve some desired state, reflected in the response.
- The `notify->consume` pattern describes the handling of some event, of which SPARKL has been notified. SPARKL eagerly looks to inform as many consumers as it can of the event occurrence, even performing intermediate operations in order to reshape the original event prior to consumption.

Essentially, the first is a synchronous pattern, and the second, an asynchronous pattern.

Figure 2 shows a possible instantiation of the `solicit->response` pattern.

SPARKL will look to satisfy the transaction, embodied as a **solicit data event** sent to SPARKL, by sequencing (request) operations on various service end-points.

The solicit has a single possible response, other than error, which is a value for the red data field. The solicit is supplied with blue and green data field values.

SPARKL sees that, in order to satisfy the solicit, it needs to sequence two operations.

Now a value for the yellow data field can be obtained given a green data field value.

Then, blue and yellow field values may be given to the `request_balance` operation, offered by a back-end service.

This service gives back a value for the red data field, which is routed back to the original caller as a response data event.

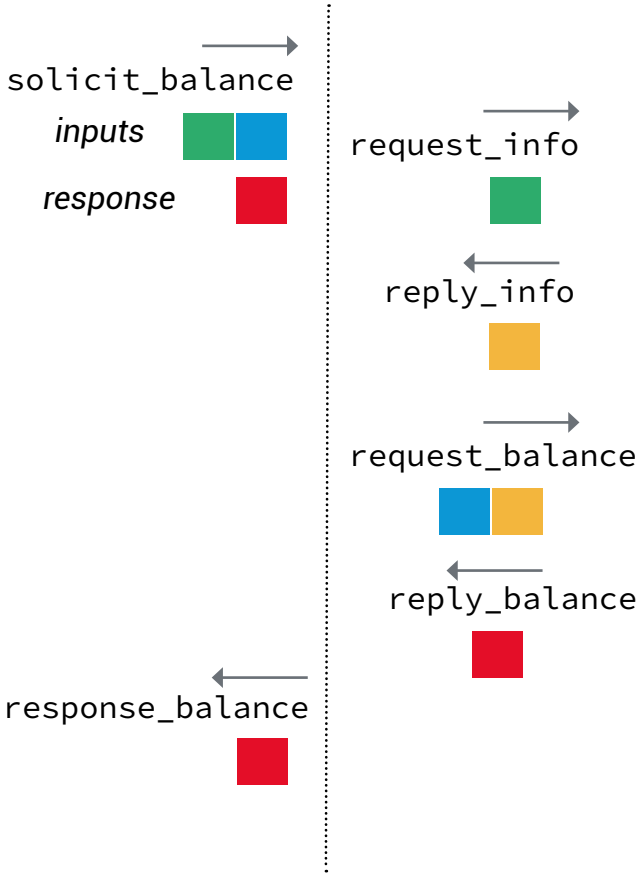


Figure 2 - SPARKL sequencing the satisfaction of a solicit transaction

SPARKL plans out the execution of operations to satisfy a solicit event, such that the operations yield sufficient data values for one of its responses.

The orchestration that takes place is captured as a mix, which is a small, self-contained definition of a **solicit** and **request operations** on services and **data fields**.

The satisfaction of the solicit transaction entails six data events which are logged for auditing purposes by SPARKL:

- the initial solicit and its response;
- and two sets of request and reply events.

Figure 3 shows a possible instantiation of the `notify->consume` pattern. When an event notification is received by SPARKL, the service contributing `consume_blue` (in the implied mix) may immediately be called. This is because the blue data field is immediately available, and is sent in a ***consume*** data event to the service.

SPARKL works out that the particular service contributing the `consume_yellow` operation may also be called if SPARKL first inserts the `reshape_green` operation in order to elicit the yellow data field.

The purpose of **request** operations, in the `notify->consume` pattern, is to reshape **notified** events prior to **consumption** by service end-points.

The `notify->consume` pattern leaches into the `solicit->response` pattern, but not vice-versa.

Consume operations may be invoked in the course of satisfying a solicit transaction to provide field data from the ongoing transaction.

This is data to be **consumed** by the service end-point for informative purposes. There is thus a `solicit->consume` aspect to the `solicit->response` pattern.

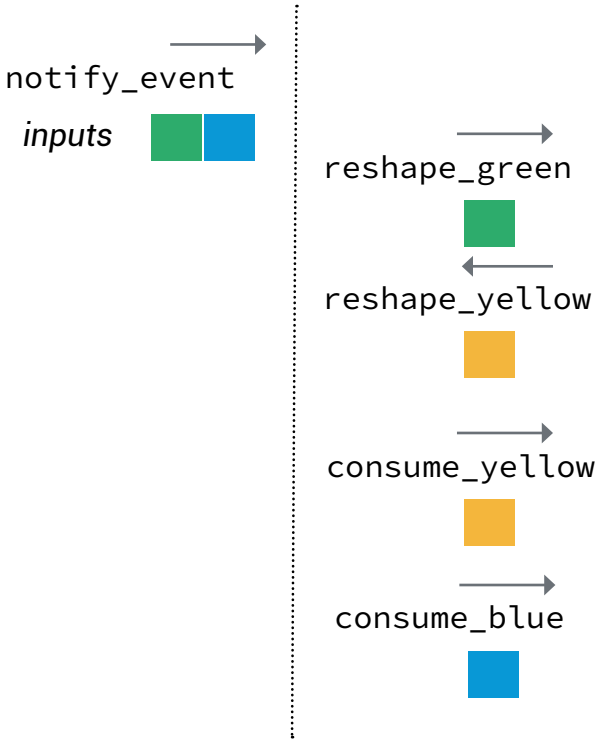


Figure 3 - SPARKL sequencing the satisfaction of a notify transaction

An ideal application of SPARKL is in the orchestration of microservices. The communication between services happens mainly according to the `notify->consume` pattern of SPARKL, promoting **loose coupling**, although the initiating call may be a `solicit->response`.

SPARKL further encourages loose coupling as its **fieldset-based semantics**, depicted above, naturally allows for changing service APIs. For instance, an operation originally may reply with red and blue data fields. If a consumer requires just red then it will still get called by SPARKL if the operation changes to returning red and green fields, as it just needs red.

It's preferable to push control logic into the services themselves, as part of the **put the smarts in the end-points** philosophy of microservices.

This helps resiliency and loose coupling. An instance of SPARKL potentially could be placed in every microservice, as depicted in **Figure 4**, to orchestrate operations and communications between processes running on the microservice, and other microservices.

Figure 4 - SPARKL-based Microservices

Given that SPARKL is baked into the microservices themselves, there is natural redundancy and resiliency in the approach. If one SPARKL is not currently available to do **load balancing** or instruct auto-scaling, for example, then another steps in. There is no need for a centralised controller.

Developers of individual microservices produce individual interface definitions as SPARKL mixes. These mixes can be arbitrarily composed to describe the behaviour of an overall system. There is no need for any glue logic - simple cut-and-paste of mix definitions to create compositions **just works**. Again, this means that the smarts are owned by the services themselves.

In a legacy setting where some of the system components will be implemented using microservices and some not, a sprinkling of SPARKL is enough to immediately see benefits with respect to capturing **data provenance** from which insights can be drawn through analytics. We offer a solution where data provenance can be captured against both non-SPARKL and SPARKL-based service components.

Data provenance is concerned with tracking the flow of data through service end-points including how it is shaped. We often want to assess this at a transactional granularity; hence, we use the term **transaction-oriented data provenance**.

There are two types of **data provenance** that SPARKL implements for microservices, both of which are highly useful. When implementing **Governance**, **Compliance** and **Risk** control mechanisms such as **Conduct Risk** assessments, there are many important questions posed around how data flows through systems (process provenance) and the sources of data used to arrive at values submitted in - e.g. compliance reports (query provenance).

Process Provenance captures the history of operations carried out on system components as well as the resulting manipulation of data records stored on systems. It captures specific events of reading and writing of such data.

- With SPARKL, process provenance provides support for the kind of drill-down event correlation described in the overview section.

Query Provenance

- The satisfaction of a query made on data stored by a system may require sub-queries to be carried out across many disparate data sources, involving one or more ETL layers, different database views or rules engines.
- Query provenance is the descriptions (in terms of meta-data) of the data records used to satisfy a query, and a graph of how the data records are combined for this purpose.
- It is therefore a statement of how, in principle, queries are satisfied rather than an account of specific queries made against a system, which would fall under process provenance.
- The meta-data of any data record may include the process provenance which details how the data record has thus far been shaped by the back-end systems.

Any system architecture, whether implemented using microservices or not, will need to support some degree of reporting along these lines. The use of SPARKL gives a mature data provenance solution, with minimal integration overhead.

In being able to support microservices that do not use SPARKL, we satisfy the criterion of not prescribing technology stacks for microservices.

In reality, an enterprise will settle on the choice of an orchestration tool, such as SPARKL, and seek to craft their microservices accordingly. It's possible that the enterprise will adopt some third party microservices, in which case a technology-agnostic approach remains important. SPARKL also delivers on that criterion.

Each microservice provides a specification of health checks (as another mix) that SPARKL should routinely apply to ensure the health of the service. In the event that a microservice instance is misbehaving according to the execution of a health check mix, SPARKL will replace it.

There are a number of technology integrations that are possible with SPARKL. One example is shown in **Figure 5**.

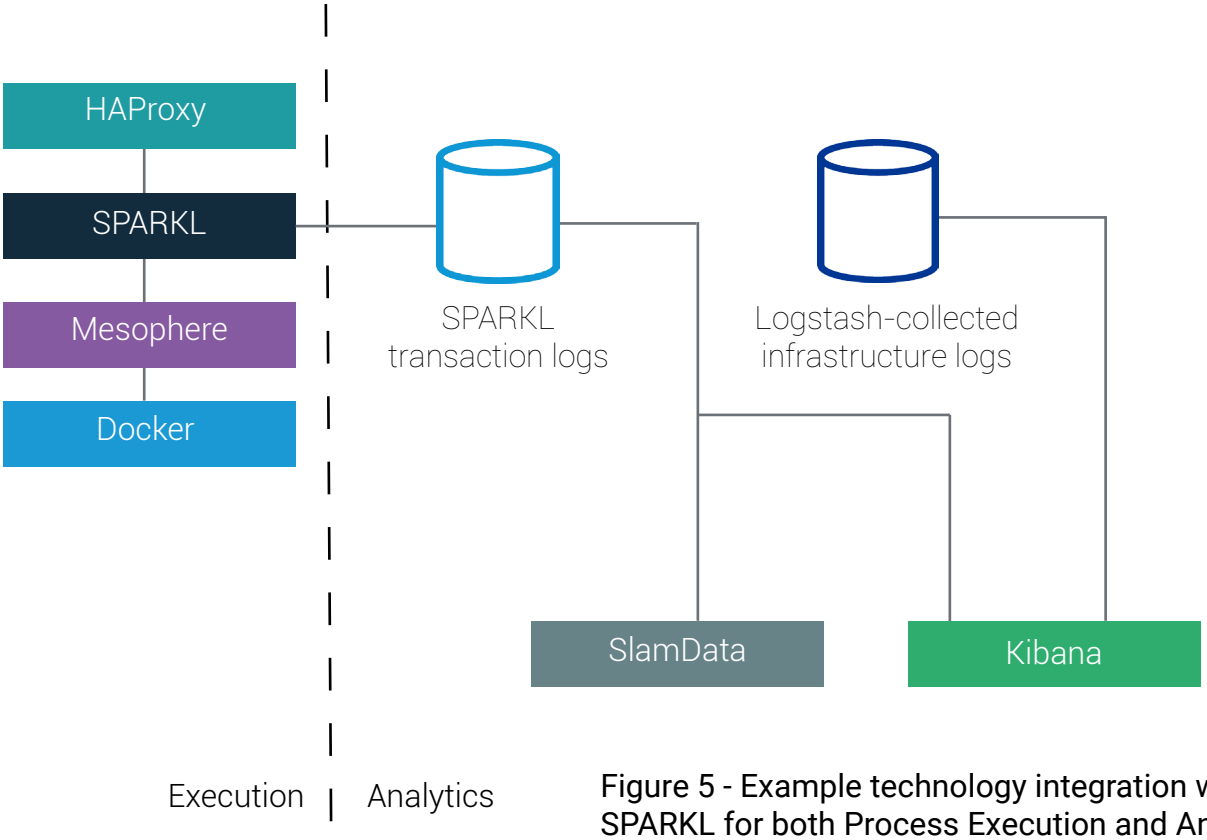


Figure 5 - Example technology integration with SPARKL for both Process Execution and Analytics with respect to this execution, including over Data Provenance trails

- **HAProxy** for high availability and initial load balancing
- **Marathon Mesophere** for docker cluster-based scheduling
- **Docker** for microservice container execution
- **Logstash** for collection of infrastructure logs from disparate environments
- **Slamdata and Kibana** for visualization, analytics and business reporting

In summary, SPARKL can be employed immediately into enterprises and start providing value, with minimal overhead:

- It is a complete legacy solution, where process-based data provenance can be extracted from the operations of existing components, as well as from new SPARKL-based microservices.
- SPARKL supports fine-grained (per-user, per-some other criterion) provisioning, load balancing and auto-scaling of microservices. It is capable of enforcing SLAs because of this.
- SPARKL supports DevOps and continuous delivery patterns typical of a microservices orchestrator, including **circuit breakers**, **green/blue deployments**, **A/B testing** and **canary releases**.
- SPARKL is scalable software, implementing a **distributed intelligence** pattern where a SPARKL router is placed into each microservices.
- SPARKL natively supports Erlang, Python and Prolog-based nanoservices.

Distributed Intelligence

Consider these examples:

- Traffic signals responding automatically when sensing an approaching ambulance to let it through congested traffic
- Determining the occurrence of a break-in or a fire in a smart building, and taking appropriate mitigations
- Real-time confidence scoring to assess in advance the potential for engine problems on an aeroplane, based on component histories

These examples are all in the realm of the **Internet of Things** (IoT), where a multitude of sensors produce a mass of data that we need to make sense of in real-time.

But by shipping sensor data off to the centralised compute facilities in the cloud, particularly when the necessary transfer bandwidth may not exist, the connection may be intermittent or have too high a latency, we cannot achieve realistic goals in IoT.

All of these examples require, to one degree or another ,a collective intelligence that isn't reliant on remote decision-making. This question belies the true **foggy** nature of IoT. IoT is about computing, **on the edge, in the fog**.

Edge computing

"Edge Computing is pushing the frontier of computing applications, data, and services away from centralized nodes to the logical extremes of a network. It enables analytics and knowledge generation to occur at the source of the data."

Fog computing

"Fog computing is an architecture that uses one or a collaborative multitude of near-user edge devices to carry out a substantial amount of storage, communication (rather than routed over the internet backbone), and control, configuration, measurement and management.

This is in contrast to these functions being carried out primarily in cloud data centres."

The principal requirement for IoT, **given its foggy nature**, is to empower controllers which are close to the source of sensor data to make sense of that data; then, to build out controller networks so that aggregate decisions can be made, and aggregated views of data are presented.

Thus, the aim is to make decisions **locally** but **hierarchically**; when a decision cannot be made locally, it is punted up the chain to the next level of aggregation.

SPARKL provides this capability seamlessly through distributed intelligence.

Through **distributed intelligence**, every SPARKL controller is endowed with a local projection of the desired system state, and the controller continually seeks to achieve that, based on its monitoring of local state and actions that it has available.

Additionally, SPARKL is able to orchestrate multiple processes at the same time, to allow for operational adaptability. For example, if a fire detection system fails, the temperature sensors for the HVAC system can switch into emergency stand-by mode and feed data into the fire detection system, ensuring safety of the building's occupants.

This would essentially be a network of sensors serving multiple purposes into which Internet of Things applications subscribe to for data. A SPARKL mix is the 'Data-as-a-Service' that gives this capability with localised analytics.

The Internet of Things landscape is rich in examples where a solution based on local and hierarchical distributed intelligence would be desirable. Some IoT contexts include:

- **Smart Cities** - Smart Parking, Smart Lighting, Smart Roads, Structural Health Monitoring, Surveillance, Emergency Response
- **Environment** - Weather Monitoring, Air Pollution Monitoring, Forest Fire Detection, River Floods Detection
- **Energy** - Smart Grids, Renewable Energy Systems, Prognostics
- **Logistics** - Routing Generation & Scheduling, Fleet Tracking, Ship Monitoring, Remote Vehicle Diagnostics
- **Agriculture** - Smart Irrigation, Greenhouse Control
- **Industry** - Machine Diagnosis & Prognosis, Indoor Air Quality & Monitoring, Offshore Installations, such as Oil Rigs and Finite State Machines
- **Smart Enterprise** - Site Services and Building Management
- **Consumer** - Connected Autonomous Vehicles, Home Automation & Insurance

By way of example, consider how SPARKL would handle the breakout of a fire in an office with several different types of sensors:

Two types of smoke detectors reporting anomalous readings of a fire trigger an alarm. SPARKL is notified locally of the fire

A **fire event** is propagated to the floor controller and onto a building controller, the appropriate level to communicate to the alarm system and set off the alarm

Human presence information for the office and surrounding floors, in aggregated form, is made available by the building controller on a SPARKL dashboard to a Site Services operator, so that the fire department will know where to concentrate their search and fire-fighting activities.

Above all, the controllers will make changes to their local environment based on local information, without decisions being made by a centralised controller.

Dot-matrix signs showing the best way out to humans that may be trapped would have their message customised by local controllers.

SPARKL at the Heart of Operations

SPARKL sits at the heart of operations, both in Enterprise Computing and the IoT. As we can see in **Figure 6**, the technology presents a spoke-like architecture where managed service artefacts hang off spokes coming from SPARKL.

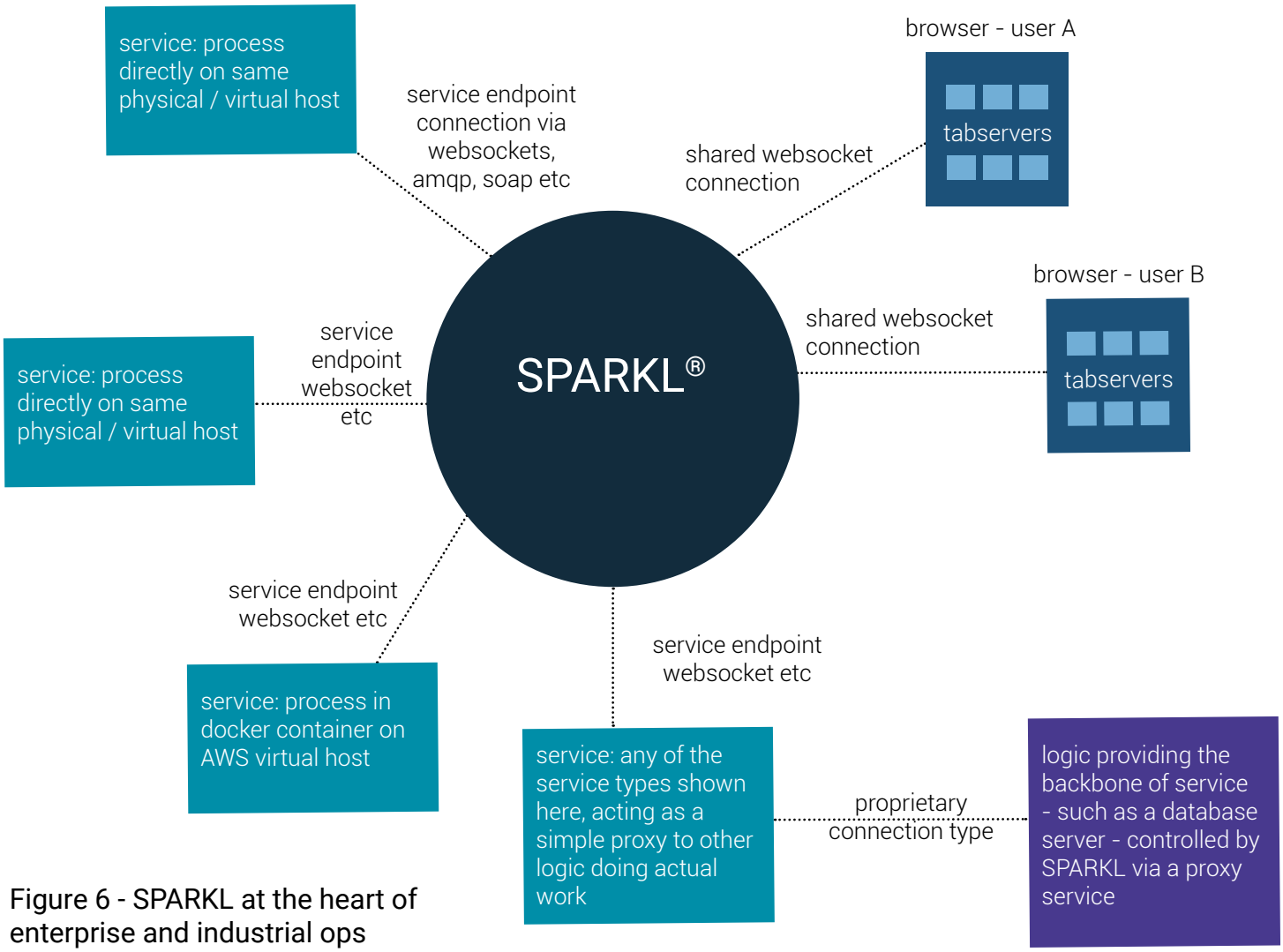


Figure 6 - SPARKL at the heart of enterprise and industrial ops

A SPARKL service artefact, or simply a service, is a compute process (e.g. a Linux process, browser thread etc) that initiates a secure websocket-based (or other) connection with SPARKL under certain conditions which are dependent on the nature of the process.

There are many ways in which a SPARKL service may be realized. In **Figure 6**, we see SPARKL services that are:

- browser-based compute threads (sharing a common websocket connection)
- Linux processes running:
  - directly on a physical/virtual machine host, or
  - within a docker container sandbox either running on the local SPARKL router machine, or on machines in AWS.

A service may also be a simple proxy routing data to and from some other compute logic, according to some non-SPARKL protocol. The example shown in the purple box in **Figure 6** is a proxy service responsible for storing and retrieving data from a database server.

There are many options. The only requirement is that the process initiates a websocket (or other) connection with SPARKL, following a specific protocol, in order to register itself with SPARKL.

Once this occurs successfully, it serves as a new service end-point that SPARKL may route data to/from, according to operations that are defined in a SPARKL **mix**, by an author of SPARKL mixes, called a **mixologist**. A SPARKL mix is an orchestration scope, limiting which operations may be sequenced in the execution of a transaction.

The data flows across the various connections, shown in **Figure 6**, are logged in a common format and are available for subsequent data analytics and mining.

SPARKL Transaction Satisfaction

SPARKL orchestration is concerned with the satisfaction of transactions. Typically, this comes down to achieving one or more goals:

- The satisfaction of a response of a Solicit operation, for the `solicit->response` pattern
- The satisfaction of the inputs to a Consume operation, for the `consume->notify` pattern.

A `solicit` operation has one or more input data fields, and one or more named Responses each with their own field sets.

A `notify` operation has an input field set, but no responses. These operations are invoked by service end-points by sending data events to SPARKL.

In the course of processing a `solicit` or `notify` event, SPARKL will invoke operations shown on the right-hand side of **Figure 8**, by sending new data events to service end-points offering these operations.

Only operations within a given mix may be sequenced together; that is Requests and Consumes may only be used to satisfy `solicit` and `notify` operations in the ***same*** mix.

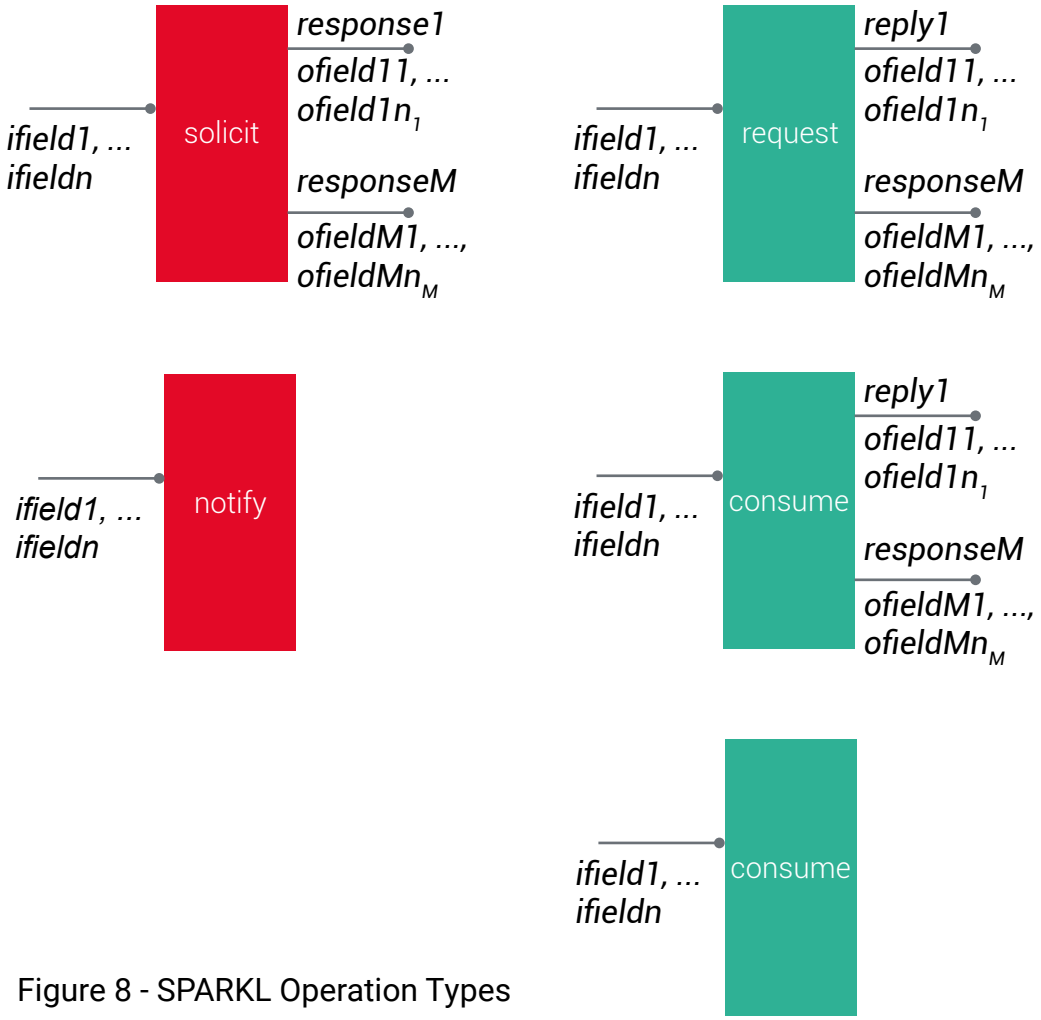


Figure 8 - SPARKL Operation Types

SPARKL sends data events to services in order to progress the satisfaction of an initial Solicit or Notify event. It determines which services should be sent events and when by using a graph-based approach to sequencing. We saw this in operation in **Figures 3** and **4**. In these figures, we represent the fields: ***ifield***$_0$, …, ***ifield***$_n$, ***ofield***$_0$, …, ***ofield***$_n$, as coloured blocks.

Consider, also, the combined picture shown in **Figure 9**. The mix shown essentially follows the `solicit->response` pattern. There are, however, `consume` operations that may be satisfied along the way. When this happens, for a `solicit` pattern, we say that there is leaching of the `notify->consume` pattern into the `solicit->response` pattern.

SPARKL will use an A* (best-cost/shortest-path) search algorithm against fieldset graphs. The graph represents the evolution of a fieldset, for a `solicit` transaction, starting with the input fieldset of ***solicit***$_A$ (shown as green and blue blocks) at the top of the graph. Each node in the graph shows a field-set that SPARKL currently **knows**.

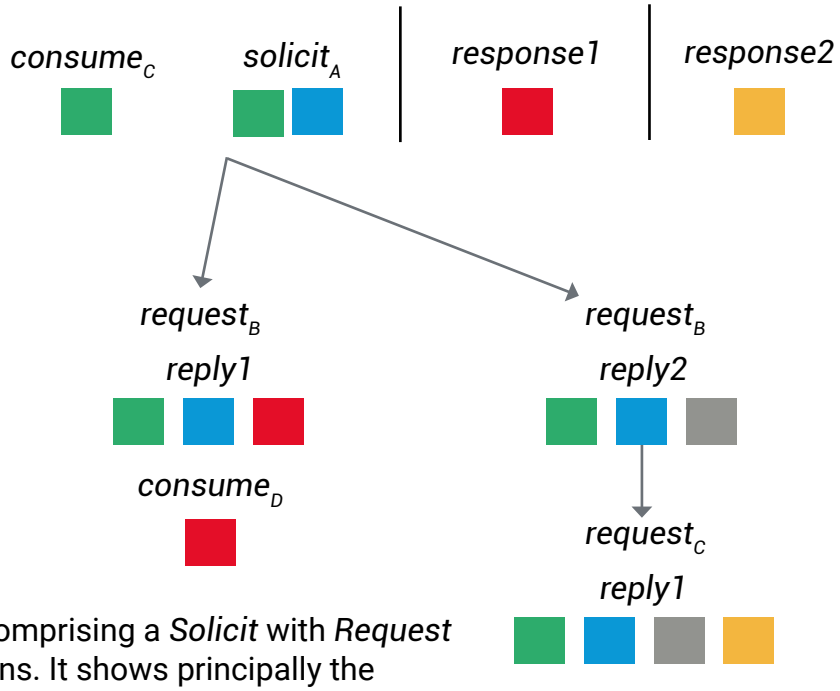


Figure 9 - Simple mix comprising a *Solicit* with *Request* and *Consume* Operations. It shows principally the `solicit-->response` pattern, with some leaching from the `consume-->notify` pattern

Request operations are the edges in the graph, and, as these are executed, the current fieldset evolves. Simply put, as we execute operations, we gather more data. In order to satisfy a `solicit` event, we need to gather sufficient data for (at least) one of its Responses.

SPARKL determines that, to satisfy ***solicit***$_A$, the ***request***$_B$ operation should be called first. This operation is first on both paths leading to the two Responses; the search algorithm will determine that it should be called because it sits first on the shortest path to a response, namely, ***response1***.

This ***requestB*** operation has two possible replies.

- If ***reply1*** is sent back (as a data event) then SPARKL now has enough field data to satisfy one of the Responses on $\boldsymbol{solicit}_A$. SPARKL sends a Response data event (with an **output** name of ***response1***) to the originating service end-point.
- If $\boldsymbol{reply}_2$ is sent back from $\boldsymbol{request}_B$ then SPARKL needs to do more work to satisfy $\boldsymbol{solicit}_A$. From the fieldset graph, SPARKL determines that it should call $\boldsymbol{request}_C$ in order to get field data that will satisfy the outstanding Solicit via ***response2***.

SPARKL sequencing is goal-driven. The two leaf nodes, in the graph shown in **Figure 9**, represent goal states in that they satisfy the Solicit operation.

Whenever SPARKL's current fieldset satisfies (i.e. is a superset of, or is the same as) the input fieldset of a **Consume** operation, SPARKL will invoke the operation (by sending a data event to the owner service end-point). A node in a fieldset graph that satisfies the input fieldset of a Consume operation is also a goal state. This is because the search algorithm will seek out these nodes.

In **Figure 9**, $\boldsymbol{consume}_C$ is immediately satisfied by the input fieldset of $\boldsymbol{solicit}_A$. A simple application of $\boldsymbol{request}_B$ may be sufficient to satisfy $\boldsymbol{consume}_D$, in the event of ***reply1*** being received from $\boldsymbol{request}_B$.

SPARKL implements an approach where, once the value of a field is known, the field continues to have that value. Thus, field data is immutable in SPARKL. There is a way to bring mutability in, and that is to jump between field-set graphs. In doing this, SPARKL creates a new thread of sequencing with a fresh slate for field data, save for those fields whose values are carried across.

Mutability is facilitated through the use of Consume operations which have Replies. The fields specified in the Reply of a Consume constitute the field data that is brought across to the fresh slate. Thus, field data is immutable in SPARKL unless it crosses a **Consume Boundary**.

In the `notify->consume` pattern, we use Consume operations as goals. Some will have replies, and when the reply occurs a new thread of execution is created. This is often handy, with this pattern, in creating multiple threads eagerly finding way in which to respond to the initial Notify operation.

Idiomatically, parallel threads of execution should not be created just to have a mutable field, and should be avoided in this case. In the case of mutability, your mix should be structured such that execution only continues in the new thread created by the **consume** and ceases in the old thread. SPARKL produces an idiom guide which makes for easy writing of mixes, following these principles.

The beauty of SPARKL is that it leaves the precise semantics for you to decide, bring a great deal of flexibility. Along the way, we provide lots of help to enable you to write powerful SPARKL mixes with ease.

Download SPARKL Now

sparkl.com
@sparkl

See SPARKL tutorials and demos at
sparkl.com/docs/web