ESAC TAP stateless

IVOA June 2025 Interoperability Meeting
Jose Osinde (Starion for ESA)
Ignacio Leon (AURORA Technology B.V.)
C. Rios (Starion for ESA, M. Henar (Starion for ESA), J. Ballester (Starion for ESA)
R. Parejo (Starion for ESA), R. Bhatawdekar (ESA)
SCO-08: Archives Software Development

ESAC
Camino Bajo del Castillo s/n, Urb. Villafranca Del Castillo
28692 Villanueva de la Cañada (Madrid) Spain

Preparing TAP+ for higher loads and complexity

TAP is becoming a fundamental component of the ESDC Archives. Looking ahead, we need to take a step forward and ensure a service that offers:

- Improved robustness
- Fault tolerance in case one of server crash
- Better support of high load peaks.
    - In a standard TAP service most of the load typically resides in the backend (the database). ESDC provides additional functionality like on-the-fly data combination and formats conversions that add significant load to the TAP server itself
- Scalability to handle more simultaneous requests from different users
- Additional flexibility to deal with new requirements

Advantages of a stateless service

A stateless service does not store any client session information between requests. Each request is independent and self-contained. This architectural approach offers several key advantages:

- Scalability. Stateless services are easier to scale horizontally (add more instances) because any instance can handle any request—no need to track session state.
- Resilience & Fault Tolerance .If a service instance fails, another can take over immediately without loss of session information.
- Better Load Balancing. Load balancers can distribute requests freely across instances without worrying about session affinity (a.k.a. "sticky sessions").
- Improved Performance. No need to read or write session data, so responses can be faster, especially in high-throughput scenarios.
- Easier Maintenance and Upgrades

TAP+ is not a stateless service

Most of the advantages offered by a stateless service are aligned with what we want for our future TAP, but TAP is not a stateless service:

- A session with associated privileges and quotes is created for any logged user
- We need to keep track of the jobs associated to a user even for the anonymous ones. This includes the job's status and any possible result
- Events/notifications and status information is cached in the server memory

Our real goal is to get a TAP service that can be integrated into a load balancing solution

Session handling

- Refactorization of the TAP library to base it in the Spring framework
- New configuration for Spring Session + Spring initialization for standardized session management.
- Session data is now persisted in a shared relational database, enabling consistent user sessions across multiple TAP instances deployed on different servers

Database access

- Transactional queries are already safe in a multi-client environment
- We now need to review queries related to Data Manipulation Language (DML)
    - This focuses on data content, not on structure (DDL), permissions (DCL), or transaction control (TCL).
- DML operations include: SELECT, **INSERT**, **UPDATE** and **DELETE**.
- Even when focusing only on DML, if we intend to share the same database across different TAP instances, we must carefully review all methods using these operations to ensure proper transaction handling where required.
- Main use cases include:
    - Register new tables in the tap schema (including table name, columns and types)
    - Job management

UWS events

This information needs now to be shared between the different instances

Jobs

- JOB_CREATED_EVENT
- JOB_UPDATED_EVENT
- JOB_REMOVED_EVENT

Login

- LOGIN_IN_EVENT
- LOGIN_OUT_EVENT

Quota

- QUOTA_DB_UPDATED_EVENT
- QUOTA_FILE_UPDATED_EVENT

Notifications

- NOTIFICATION_CREATED_EVENT
- NOTIFICATION_REMOVED_EVENT

Sharing

- SHARE_ITEMS_CREATED_EVENT
- SHARE_ITEMS_UPDATED_EVENT
- SHARE_ITEMS_REMOVED_EVENT
- SHARE_GROUPS_CREATED_EVENT
- SHARE_GROUPS_UPDATED_EVENT
- SHARE_GROUPS_REMOVED_EVENT
- SHARE_USERS_CREATED_EVENT
- SHARE_USERS_UPDATED_EVENT
- SHARE_USERS_REMOVED_EVENT

Service start/stop I

A TAP service must now consider the possibility of being configured with multiple instances.

- Startup clean-up procedures need to be redefined to support this setup:
  - How many instances should run in parallel
  - Can this be adjusted in real time? How does this impact the overall service?
  - How should pending jobs be handled? Who is responsible for cleaning them up, and when?
- Which model should we choose: Master/slave architecture or a decentralized one
- At the same time, we must maintain backward compatibility with archives running on a single-server configuration.

Service start/stop II

On TAP startup:

- Create new uuid for this instance
- Clean up uuids table: remove 'inactive' rows
  - Inactive == without a 'recent' heartbeat
- Execute 'Restart Job Procedure' for:
  - All other jobs with 'inactive' AND non-existent uuids (they became inactive and/or were removed)
    - **Risk:** What if several instances start at the same time and do this last point concurrently? To avoid that, do the following:
    - LOOP (execute the things below transactionally!)
      - Change job's uuid to be the new one (so that nobody else takes it)
      - Restart it / Cancel it / Delete it
      - Wait x min
      - take next one (check if still exist)

Generic changes:

- When creating new job, add uuid (new column 'owner_instance') to it
- When changing the phase/status of the job, update its uuid (owner_instance) to reflect which instance is in charge of that phase.

System tests

The architecture is now more complex, and so are the test required to cover the different use cases. We are now dealing with issues similar to those encountered in multi-thread environments:

- The setup must launch multiple instances in parallel
- We need to test interactions between two or more TAP instances sharing the same database. Typical use cases include:
    - Ensure that tables registered in one instance are visible to the others
    - Retrieving and manage jobs created from different instances
    - Handling shared resources (tables, results), and more
- Race conditions must be carefully considered and tested
- Another non-trivial scenario is to testing the service during startup and shutdown sequences

Questions / feedback

Thank you for your attention