2. As Tigran mentioned, over its 20-year lifetime, dCache has acquired a broad spectrum of users, with different workflows, access, capacity and data preservation requirements, resources and budgets. This means there can be appreciable differences in the desires and expectations regarding how data is stored and made available –– what we are calling “Quality of Service” or QoS for short. I will be speaking today briefly about one of the steps we have taken to enhance dCache’s capacity to respond to these QoS needs.

3. There are numerous modes for deploying dCache––I would guess almost as many as there are sites using it; but two of the most basic ways are summarized by this slide: as a disk cache for archival/tertiary storage, or as a disk-only system with multiple permanent disk replicas of the data. These modes can also be combined (such as with DESY Cloud).

4. In order to realize these basic modes of provisioning data, then, dCache must be able to flush data to HSM systems, writing to disk one or more cached copies which can be purged and then restored from the backend; and it must be able to maintain multiple persistent replicas on disk. The latter capability is what we refer to as “[data] resilience”.

5. The dCache subsystem called the Resilience service or simply “Resilience” is responsible for achieving data durability (preventing data loss) for disk-only deployments. Within a given installation, the pool groups and storage units requiring resilient management are specifically marked as such (in other words, Resilience rests on an explicit partitioning of the dCache configuration). Now, from the perspective of QoS, Resilience looks not only like a way to achieve data durability, but also data accessibility: that is, keeping certain classes of data on disk for faster access.

6. And in fact, much of the underlying concept and design of the Resilience service can be generalized to the include other QoS transitions. Its current implementation, however, prevents easy extension in this direction. To achieve this, we need to remove the partitioning of resilient from non-resilient data, and pry apart several layers into clearly distinguishable components or APIs, in order then to be able to place on top of this a rule engine for defining how sets of data or even individual files need to be treated.

7. The bird’s-eye-view at the top of this slide represents the main components of Resilience responsible (on the left) for processing individual files and (on the right) for checking the consistency of data on a pool-by-pool basis when changes of state occur (pools go offline or come back online). All of this has been fairly tightly coupled (for reasons of optimization), but over the course of several years of experience running Resilience, we discovered a few issues which required breaking some of the original constraints, so that now it makes perfect sense to go ahead and separate out the main components into independent services.

8. This slide shows how this adaptation was accomplished. The new components are named in order to reflect well-defined functions: receiving messages, providing requirements, verifying if the requirements are met, making adjustments if they are not, and scanning the system to ensure consistency.

9. These can now run as separate services in one or more dCache domains. The Engine, or entry point, responds to messages from the namespace concerning new cache locations, and to RESTful requests for changes to QoS class either singly or in bulk, while the scanner handles changes in pool status and configuration from the pool manager.

10. The two most salient achievements of this refactoring are: First, isolation of the parts of Resilience/QoS which delegate to other dCache components the heavy lifting such a copying, flushing, restoring, and caching of files (the Adjuster) –– this will become particularly important when Lea’s work on restore optimization gets fully integrated; and second, the accommodation, through a high-level API, of new ways to define and manage QoS requirements for individual files (the Provider or “Rule Engine”).

11. As a first step in the latter direction, we have prototyped this Provider using the current way of determining file QoS: by a combination of namespace attributes having to do with accessibility (Access Latency) and durability (Retention Policy), and storage unit attributes prescribing the number and distribution of permanent disk copies.

12. This results in the following mapping to four broad QoS classes: volatile (temporary, scratch-like data), tape (not guaranteed to be on disk), disk (but not on tape) and disk + tape.

13. On the basis of these classes, we have made available a limited set of transitions between them which can be triggered via a RESTful API (implemented by the dCache Frontend and Bulk services).

14. Here you can see that the request transition triggers a change in the file attributes in the namespace along with the necessary copying, caching, flushing to the backend or restoring to disk. The most useful of these transitions for most experiments I would imagine is the **tape => disk + tape** for files not currently on disk (third from bottom); that is, what is normally called bringonline, stage or pre-stage.

15. This prototype is, of course, limited. Only the namespace attributes can be changed for individual files; their number of copies still depends on a global, static storage unit definition which can only be changed through system configuration. Also, it does not support storage on multiple tertiary systems at the same time. Finally, and most significantly, there is no way of expressing automated changes triggered according to a set of rules or conditions, possibly temporal. **Adding these features will be the next steps in dCache QoS development.**