P2ware expands services and speeds delivery using Azure Service Fabric

April 26, 2018

Authored by Marcin Kosieradzki, CTO at P2ware, in conjunction with Ed Price and Mark Fussell from Microsoft.

This is part of a series (https://blogs.msdn.microsoft.com/azureservicefabric/tag/case-study/) on customers, where we look at why they chose Azure Service Fabric (https://azure.microsoft.com/en-us/services/service-fabric/) , and dive deeper into the design of their application, particularly from a microservices perspective. In this post, we profile P2ware, their project and portfolio management web applications that are running on Microsoft Azure, and how they designed the architecture using Service Fabric.

P2ware (https://p2ware.com/) has been providing project management software solutions since 2004. Our mission is to provide highly customizable and affordable enterprise project and portfolio management solutions to the midmarket customers. Previously, most of our solutions were deployed on-premises.

Cloud computing is the key element in our long-term strategy, as it not only allows us to expand our business and provide our services to a much wider audience, but it also allows us to shorten our deployment and customization cycles drastically. As a result, we can apply an even more agile process and

eliminate risk for our customers. This is achieved through a completely new deployment model—cloud-based prototyping.

We design our systems in a way that enables our customers to move their applications and data to an on-premises datacenter or to a private cloud provider. The challenge is to achieve feature parity between on-premises solutions and cloud solutions.

Data model

One of the biggest challenges in the pre-Service Fabric era was our complex data model. It consists of two core-concepts: plan and organization.

Plan

Our data model includes thousands of plans per tenant. A *plan* is a core building block of the P2ware systems. Each plan has its individual data model and schema, user interface customization (including advanced interactive diagrams), data grids, and complex input forms. It can be highly customized using our business rules engines, including formulas, constraints, triggers, and stored procedures and functions.

A plan is like a microdatabase. It's a core container for all data that's related to a single project, portfolio, organization unit, or another large entity. Plans have access control lists and can be synchronized between a server and a replica run by a desktop client application, in order to enable offline work.

Organization

Plans are connected across an organization using different highly specialized workflows that enable the following:

- Integrating data between plans.
- External system integration.
- Building customer-specific workflows (including approval processes, notifications, and so on).
- Keeping plan-based schemas and customization up to the organization's standards and requirements.

All users, processes, and customization are specific to an organization (which should be considered as a single tenant).

“ Service Fabric is a critical element and enabler in our cloud strategy.”

—Marcin Kosieradzki: CTO
P2ware

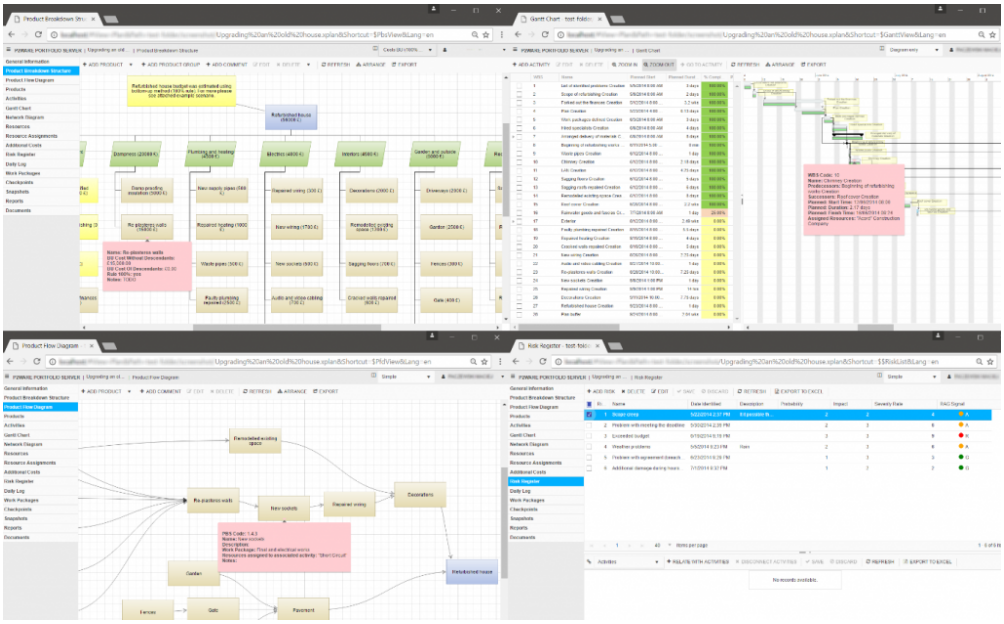


Figure 1: P2ware cloud screenshots

Previous architecture

Historically, we tried many different approaches to store and process our plan data (see Figure 2). Our previous best solution included the following two tiers:

- **P2ware Portfolio Server 2011** – Our front-end web application was hosted by Internet Information Services (IIS). It was doing a lot of memory-heavy caching.
- **Microsoft SQL Server database** – We had two advanced features enabled: FILESTREAM (used to store blobs efficiently) and Service Broker (used as a message bus that facilitated cache invalidation and sent reliable notifications to external systems).

It was much more efficient than the many different solutions we tried earlier, but it had the following downsides:

- In the more pessimistic scenarios, each application potentially needed to cache all the plans in memory (this was causing large memory requirements).
- Cache misses were introducing additional latency and CPU load. Every cache miss requires you to reload data from the blob storage. That problem was partially mitigated by the load balancer sticky sessions.
- Optimistic locking was responsible for additional work under heavy multi-user activity on a single plan (such as rollback and retry).
- The database required some maintenance and administration.

This architecture was definitely not cloud-ready nor efficient enough.

> " Service Fabric provides us with a great easy to deploy, easy to manage reliable actor experience. This actor model element of Service Fabric alone would be a good reason to adopt this technology, but that's only the tip of the iceberg! "
>
> —Marcin Kosieradzki: CTO
> P2ware

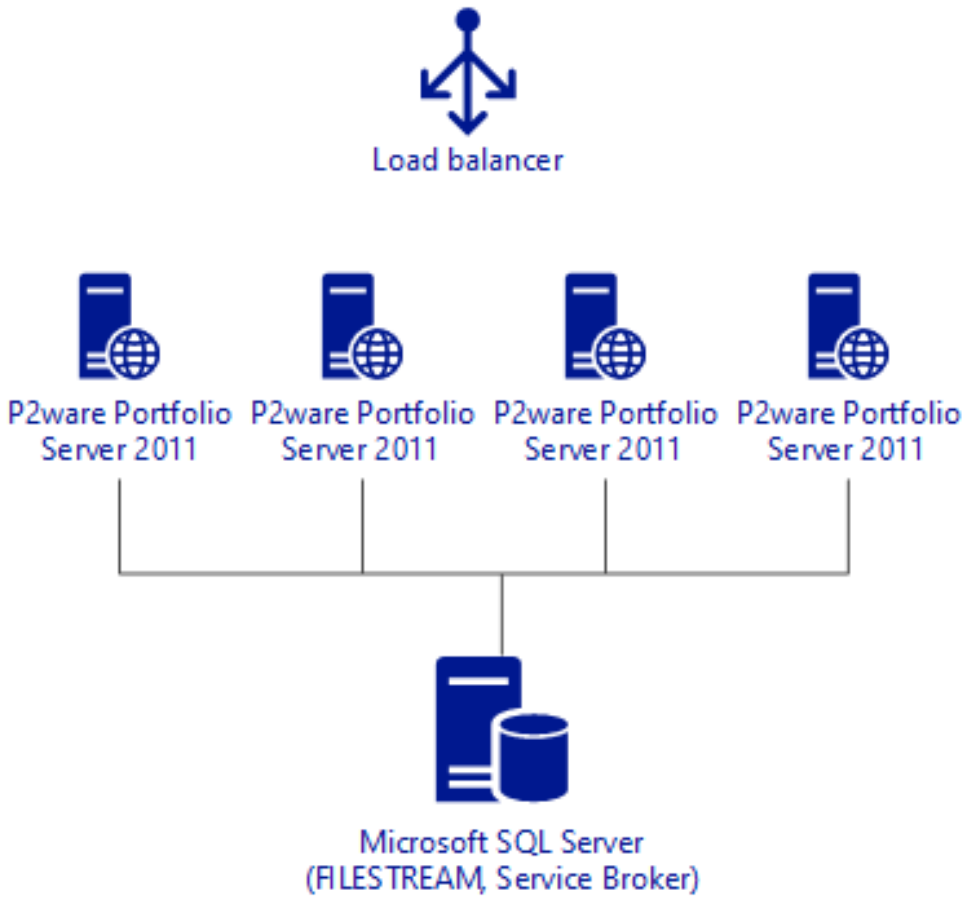


Figure 2: Previous project management architecture

Current architecture and the actor model

As a result of a long R&D investigation, we decided to use the actor model approach. It turned out to be a perfect fit for our data processing challenges.

The following are the benefits of an actor model approach, where we model a plan as an actor:

- It easily scales to running thousands of plans per tenant, which are distributed across all the cluster nodes.
- It allows you to query multiple plans in parallel, which gets you results simultaneously. The more nodes there are in a cluster, the more efficiently the system works.
- It allows you to keep actively queried plans in memory, which reduces latency as much as possible.
- Each plan is only loaded by one node, which drastically decreases memory usage in scale-out deployments.
- Turn-based concurrency ensures the data is consistent, and it drastically increases throughput when compared to optimistic locking approaches.
- It ensures the best resiliency. In case of an application or node crash, the plan is reloaded from the stateful Organization Service (see Figure 3), which introduces only a slight delay.
- Data is loaded into memory on demand, which means that rarely accessed data is not a burden to the system.
- In the event of high memory pressure, we can provision additional machines by using the autoscaling feature of Azure Virtual Machine Scale Sets (https://azure.microsoft.com/en-us/services/virtual-machine-scale-sets/) (VMSS).
- We can trade memory for processing power and I/O usage by controlling actor unloading policies.

Service Fabric at the organization level

Scalable and efficient plan data processing is the first problem solved by the new architecture. The second problem solved by Service Fabric is running a partitioned, stateful service that is responsible for handling an organization. All the organization-level data is stored in-process in a cluster-replicated Reliable Collections

that guarantees nearly zero latency (in-memory) in accessing all the key value dictionaries and queues. This completely eliminates any reason for implementing an additional caching layer—all the required data is right here, right now, and it is durable.

We find the Reliable Services programming model to be very productive (especially when compared to the relational database) and comprehensible. Organizations are distributed across cluster nodes, using a partition mechanism to ensure optimal scalability.

We decided to handle HTTP requests using ASP.NET Core inside the appropriate stateful Organization Service, which is optimal from the latency perspective, and it also removes any need for implementing an additional communication layer. The only exceptions include requests that should be processed by Plan Actors. In that case, the appropriate part of the request is sent to the Plan Actor Service, by using Service Fabric remoting (see Figure 3).

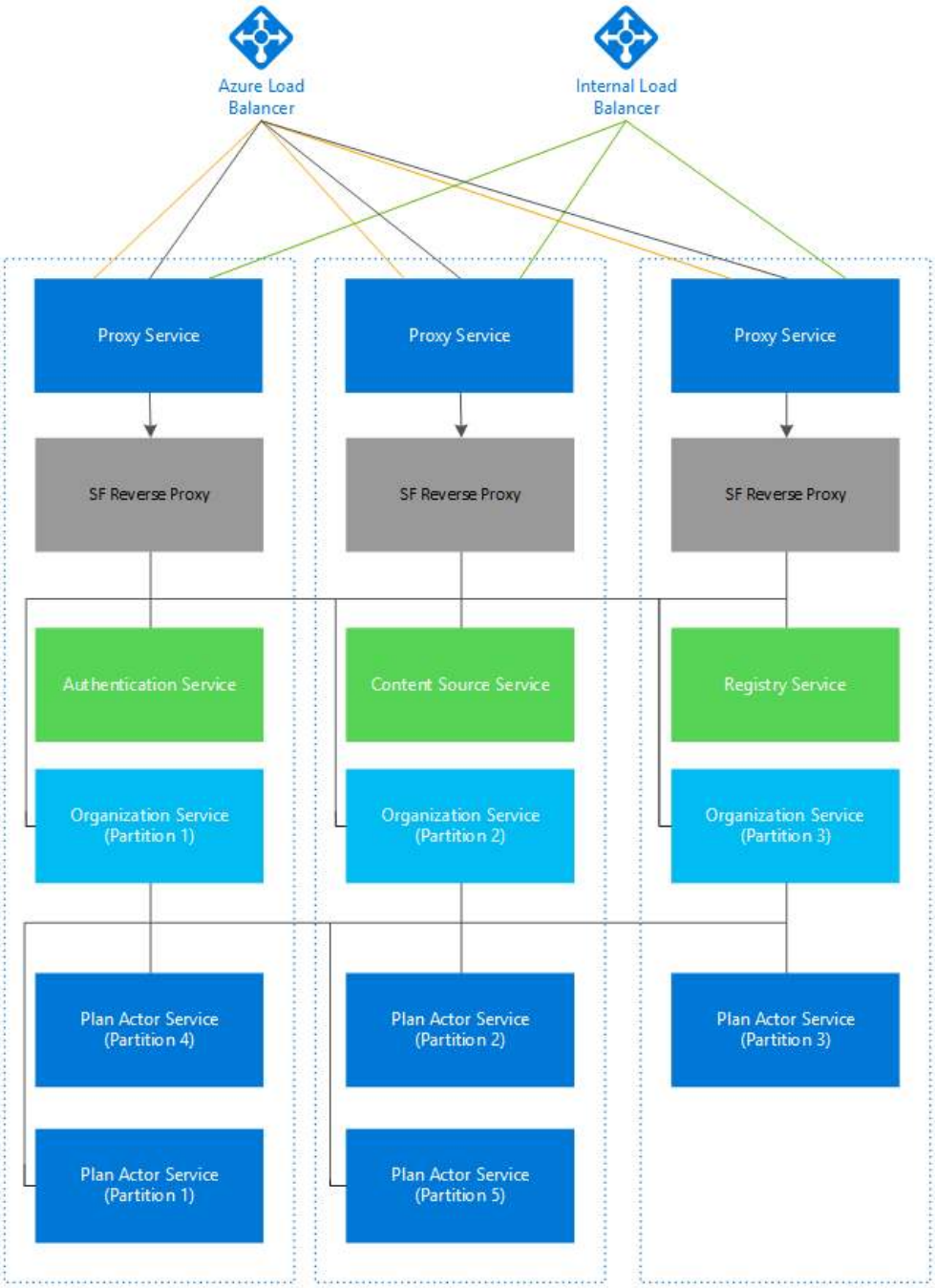
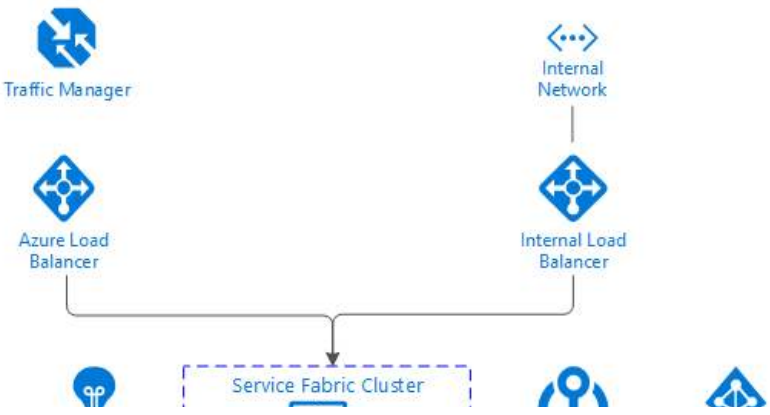


Figure 3: The core services architecture



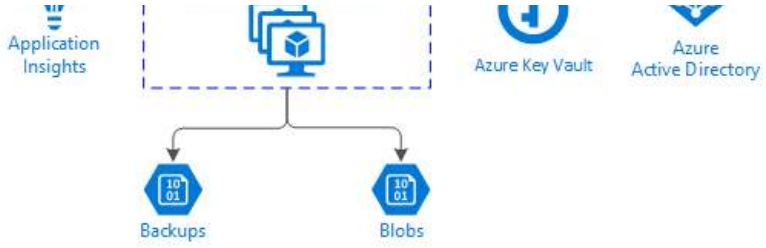


Figure 4: Azure services

HTTP services

Like in the case of our Organization Service, we decided to use HTTP as the main transport for all our services. Figure 3 shows all our core-services:

- **Plan Actor Service** and **Organization Service** have been described.
- **Authentication Service** – Responsible for authentication by using Azure Active Directory (https://azure.microsoft.com/en-us/services/active-directory/) , Azure Active Directory B2C (https://azure.microsoft.com/en-us/services/active-directory-b2c/) , or any other customer-required identity provider.
- **Content Source Service** – Provides up to date static content, including HTML and ClickOnce applications that are intended to be cached higher in the hierarchy or in a CDN, like Azure Content Delivery Network (https://azure.microsoft.com/en-us/services/cdn/) .
- **Registry Service** – A global service that helps resolve an organization name to an API URL that is responsible for the specific organization. This is a future-proof design that allows us to move a specific organization to a different cluster when needed, without disrupting the user experience.
- **Proxy Service** – A configuration-driven proxy that is responsible for translating addresses in a Service Fabric reverse proxy–friendly manner. It provides multidomain support and security-related facilities, like throttling and service isolation between Azure Load Balancer (https://azure.microsoft.com/en-us/services/load-balancer/) and the internal load balancer. For an additional layer of security, some administrative services are only accessible by using an internal load balancer. Most of the heavy lifting (service resolution and basic retry handling) is done by the Service Fabric Reverse Proxy.

Azure services

One of the most important capabilities to our architecture are standalone clusters, which means that a Service Fabric application can be deployed and run with complete feature parity on-premises. This is also

the reason why our design tries to limit the use of additional Azure Services that might be not available at our customer site.

Nonetheless, we are using the following Azure services (see Figure 4):

- **Traffic Manager** (https://azure.microsoft.com/en-us/services/traffic-manager/) – First tier traffic routing allows immediate traffic redirection when required.
- **Azure Load Balancer** – Distributes incoming traffic to our proxy services.
- **Internal Load Balancer** together with **Azure Virtual Networks** (https://azure.microsoft.com/en-us/services/virtual-network/) – Enables internal connectivity from our sites.
- **Application Insights** – Helps us gather additional telemetry.
- **Azure Key Vault** (https://azure.microsoft.com/en-us/services/key-vault/) – Enables us to manage secrets.
- **Azure Active Directory** and **Azure Active Directory B2C** –Provides us with identity management for an ultra-wide range of scenarios.
- **Azure Blob Storage** (https://azure.microsoft.com/en-us/services/storage/blobs/) – Used for backups and for storing large and rarely accessed blobs (to offload data stored in Reliable Services collections).
- **Azure-managed Service Fabric** – Built on top of Virtual Machine Scale Sets (VMSS) it offers superior resiliency and autoscaling.
- **Azure DevOps (formerly Visual Studio Team Services)** (https://azure.microsoft.com/services/devops/) – Enables us to use a fully integrated continuous integration and deployment (CI/CD) pipeline.

Summary

Service Fabric has allowed us to design a comprehensible system that can greatly benefit from being hosted on Azure, but it is also easily migratable to the customer site on-premises.

It's particularly outstanding when compared to our previous monolithic, classical SQL database-based design:

- It's self-healing, and is designed to work with or without limited human intervention. Instead of doing maintenance chores, we are focusing on writing reliable self-maintenance tasks that utilize proven procedures. Using automated deployment together with proper procedures helps us avoid human-errors.
- It allows us to achieve 10x better Azure resource utilization (memory and CPU) and density. Thanks to that, we can deliver solutions that are more reliable and affordable, at the same time.
- The average request latency has been decreased by orders of magnitude to an unnoticeable level, which greatly improves the user experience.
- Instead of depending on multiple third-party services, we are depending only on one service, which has been battle tested by running Microsoft's core Azure services. As a side effect, we save on third-party

license fees, which improves profitability.

- The new tenant provisioning cost has dropped to nearly zero. That allows us to provide demo versions and focus on what's most important: the value added.

> " A stateful microservices-based architecture (which includes reliable actors) is much more predictable and far less error prone. We can focus on the value added instead of dealing with problems like cache invalidation."
>
> —Marcin Kosieradzki: CTO
> P2ware