UNIVERSIDAD
DE GRANADA

TRABAJO FIN DE GRADO

Doble grado en Ingeniería Informática y Matemáticas

MW Store: facilitador genérico para proveedores de servicios con multi-inquilinato en una infraestructura Cloud

Autor

Rubén Morales Pérez

Director

Manuel Isidoro Capel Tuñón

Departamento de Lenguajes y Sistemas Informáticos

Originality statement

Rubén Morales Pérez

I explicitly declare that the work presented as TFG, corresponding to the academic year, is original, in the sense that no sources have been used for the preparation of the work without proper citation.

Granada September 3, 2022

Signed:

D. Manuel I. Capel (tutor), Profesor del Área de Lenguajes y Sistemas Informáticos del Departamento LSI de la Universidad de Granada. Informa:

Que el presente trabajo, titulado "MW Store: facilitador genérico para proveedores de servicios con multi-inquilinato en una infraestructura Cloud", ha sido realizado bajo su supervisión por Rubén Morales Pérez, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda. Y para que conste, expido y firmo el presente informe.

Granada September 3, 2022

Signed:

MW Store: facilitador genérico para proveedores de servicios con multi-inquilinato en una infraestructura Cloud

Palabras clave

Multi-tenencia, Concurrencia, Redes Petri Coloreadas, Modelos Matemáticos, Calidad de Servicio, Nube.

Resumen

La tendencia en informática hacia una externalización a la nube incentiva la adopción de ciertas prácticas en el desarrollo del software [32]. La nube como concepto permite aumentar la seguridad informática gracias a la restricción de acciones y/o permisos a los que los usuarios tienen acceso. La multi-tenencia reduce el coste del software compartiendo recursos como los servidores o las bases de datos entre los diferentes usuarios. Hay varias propiedades deseables para garantizar buena calidad de servicio, como son la escalabilidad, elasticidad y tolerancia a fallos. A la hora de gestionar la infraestructura para almacenar los datos hay varias consideraciones a tener en cuenta. Los tenants pueden requerir configuración personalizada, scalabilidad de datos con réplicas y particiones, consistencia de datos, escalabilidad horizontal dinámica para responder ante aumentos de carga en el sistema. Es necesario elegir entre la completa disponibilidad o consistencia de los datos. Dependiendo de los requisitos funcionales del sistema las propiedades ACID y BASE influenciarán la elección de la base de datos adecuada. La complejidad creciente del software requiere cierto esfuerzo para poder controlarla de forma efectiva. Cualquier sistema reactivo (con o sin bases de datos) se puede modelar usando redes de Petri coloreadas (CPNs). Este modelado matemático de los sistemas permite demostraciones formales de características deseables en aplicaciones del mundo real. Ejemplos de dichas caracaterísticas son la disponibilidad de recursos, vivacidad y seguridad. Las CPNs permiten entender las limitaciones de concurrencia de nuestros sistemas. De esta forma se pueden diseñar sistemas capaces de resistir ingentes cantidades de peticiones por segundo. Este proyecto permite a las empresas crear una web con sus productos. Los clientes de dichas empresas serán usuarios finales del sistema y podrán comprar los productos online en la página correspondiente de cada empresa. MW-Store está modelado usando el software CPN Tools, un sistema gratuito disponible balo la licencia GNU General Public License (GPL) version 2. CPN Tools ha ayudo a demostrar la correctitud del diseño de MW-Store verificando las propiedades deseables del sistema. De esta forma, se han extraído conclusiones sobre la calidad de servicio del sistema implementado.

MW Store: generic handler for multi-tenant service providers in a cloud infrastructure

Keywords

Multi-tenancy, Concurrency, Colored Petri Nets, Mathematical Modeling, Quality Of Service, Cloud.

Abstract

The trend toward outsourcing computing in the Cloud encourages the adoption of certain practices in software development. The Cloud as a concept makes it possible to increase IT security by restricting the actions and/or permissions to which end users have access. multi-tenancy reduces the cost of software by sharing resources such as servers or databases among different users. There are relevant properties for adequate quality of service, like scalability, elasticity, and fault tolerance. When it comes to data persistence, there are some trade-offs to consider. Tenants might require custom extensibility, data scalability with replications and partitions, data consistency, and dynamic horizontal scalability to respond to increasing demands, which means transitorily changing the system's load. It is not possible to have both availability/latency and consistency in distributed data stores [15]. Depending on the functional requirements of a system, ACID or BASE properties will influence the initial decision of tenants' data store(s) implementation. The increasing software complexity requires effort to manage it effectively. Any reactive system (with or without a data store(s)) is modelable with Colored Petri Nets (CPNs). This mathematical modeling of systems allows us to perform formal demonstrations of desirable features in real-world applications, e.g., resource availability, liveness and security. CPNs allow us to understand the concurrency limitations of our systems. In this way, systems can be designed to withstand huge amounts of requests per second. The project allows companies to create a website with their products. The customers of each tenant will be able to purchase those products online at the corresponding website. MW-Store is modeled using the software provided by CPN Tools, a software under the GNU General Public License (GPL) version 2. CPN Tools has assisted us in demonstrating the correctness of the MW-Store design by verifying the desirable properties and thus being able to draw conclusions about the quality of service of the implemented system.

Contents

1 Introduction

Modern society's demands explain the tendency towards more complex and configurable software. Configurability extends a system's functionality, including the possibility of simplifying the interfaces to handle just the specific requirements of each client. Internet evolution as the standard communication network facilitates its adoption as an abstraction layer over the operating system. Computation outsourcing to the Cloud has several advantages like increased security, no local installation, longer battery life, etc. [11]. The Cloud also creates a dependency on the Internet that might not be acceptable for all critical systems. Virtualization is essential to offer web services on demand, improving resource efficiency and reducing overall business costs.

Cloud Computing and Software-as-a-Service (SaaS) influence is growing in enterprise software development. SaaS is usually handled using multi-tenant paradigms. The formal definition of multi-tenancy (from [12]) is:

Definition 1. **Multi-tenancy** "property of a system where multiple customers, so-called tenants, transparently share the system's resources, such as services, applications, databases, or hardware, with the aim of lowering costs, while still being able to exclusively configure the system to the needs of the tenant." [12]

Each tenant handles its own customers' data. In general, it is not possible to access data from other tenants' directly [12].

Multi-tenancy helps to handle just one system to serve several clients at the same time [23]. This approach is also helpful in data layer access since it allows database management systems to handle several tenants with uneven requirements in just one place. Sharing data access layers helps to overall scalability. The main drawback is that it makes tenants services dependent on the load of other tenants. This dependency is handled with some scalability approaches.

Multi-tenancy still allows customization of the system behaviour to meet different customer requirements. Indeed, incremental software development approaches combined with tenant configuration allow custom interfaces for diverse clients. Even within the same field, some clients may need complex interfaces and requirements, while others may prefer simplicity. From an economic point of view, this customization facilitates remarkably different bills adapted to the client's budget. Multi-tenancy can be applied to several levels in a system's architecture and there are different architectural approaches to define it. It is possible to refer to multi-tenancy by using the whole picture of a system or by increasing the granularity to refer to the database level. To properly use the term multi-tenancy, several core principles have to be covered: data isolation, scalability, concurrent data access, performance and security [32].

Definition 2. **NIST definition of Cloud Computing.** "Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.” [32]

Infrastructure outsourcing provided by Cloud Computing tends to reduce risks like hardware failures, and becomes responsibility of of the infrastructure providers [32]. Virtualization allows Cloud Computing to pool computing resources and distribute them on demand to the applications. Resources, like software, are now deployed off premise. Dynamic resource provisioning can be either reactive, i.e. responsive to sudden demand variations; or proactive, allocating resources before they are needed by using demand forecast methods [32]. Cloud Computing architecture (Figure 1) could be described in four layers [32]:

- Hardware: handles the cloud physical resources [32].
- Infrastructure/virtualization: pool of storage and computing resources, e.g. virtual machines [32].
- Platform: operating systems and application frameworks [32]. It tries to reduce the effort to deploy applications. Platform as a Service (PaaS) provides these resources on demand.
- Application: the Cloud applications with Web interfaces [32].

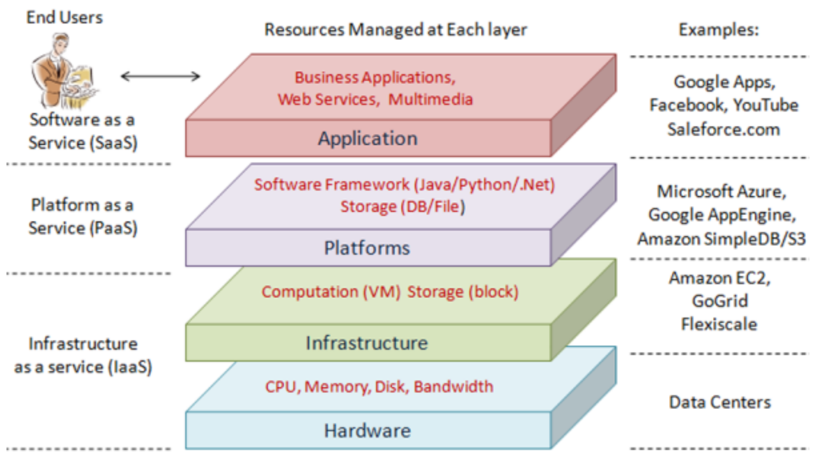

End Users
Resources Managed at Each layer
Examples:
Software as a Service (SaaS)
Business Applications, Web Services, Multimedia
Application
Google Apps, Facebook, YouTube Saleforce.com
Platform as a Service (PaaS)
Software Framework (Java/Python/.Net) Storage (DB/File)
Platforms
Microsoft Azure, Google AppEngine, Amazon SimpleDB/S3
Computation (VM) Storage (block)
Infrastructure
Amazon EC2, GoGrid Flexiscale
Infrastructure as a service (IaaS)
CPU, Memory, Disk, Bandwidth
Hardware
Data Centers

Figure 1: Cloud computing architecture [32].

From an economic perspective, economy-of-scale is easier to achieve by implementing Cloud-based solutions in huge data centers, which leads to better manageability [32]. A disadvantage of this approach is the energy consumption and initial investments [32], although the relative cost per server will hopefully be reduced or the necessary computing resources located in centers with lower electricity costs. On the other hand, smaller data centers will not require that amount of power consumption and could be preferable in case of tight budget. To prevent single points of failure and increase potential availability several smaller globally distributed data centers can be considered.

1.1 Classification

From the isolated approaches to more shared data [3], multi-tenant data classification is a continuum:

- Separate databases. Data isolation is easier to achieve using this approach. Generally, the application will execute the same code in the same server(s) for all the tenants, but this extra data security layer helps in data isolation. The main disadvantage is handling several connections to different databases, leading to higher maintainability costs. There is a limit on the number of databases that a single server can handle, a constraint that is not that important in other approaches.
- Shared Database, Separate Schemas. Database tables for each customer are grouped into a separated schema. This approach shifts security to the database permissions layer (like in the separate databases' scenario), and it is cheaper than separate databases [32]. It is the most used due to its trade-offs concerning resource usage, performance, and security.
- Shared Database, Shared Schema. Database tables store records from multiple tenants. A table column links each record with the appropriate tenant. This approach is the cheapest one, allowing the largest number of tenants per server. The higher number of rows per table is a limiting factor that could be reduced using indexes at database level. One disadvantage of this approach is that when customer requirements are very different, there is a performance overhead accessing the database. This overhead will be analyzed with the different extensibility patterns for multi-tenancy.

Some customers will have stronger data isolation requirements than others [32]. Separate databases add an extra security layer due to more granularity in database access permissions. Your business could take the decision based on expected requirements like the number of tenants, data per tenant, tailored tenant services, etc. Separate databases or schemas would allow easier configurability per tenant, creating custom tables and columns as needed. The business model is relevant here because more shared approaches would make feature toggle straightforward, allowing turning on/off certain features on the fly.

On the other hand, just because a system is advertised as SaaS in the cloud does not mean it holds its benefits (savings, security, flexibility, disaster recovery, scalability, access to automatic updates, ...) [13]. Microsoft describes an incremental SaaS maturity along three dimensions: configurability, multi-tenancy and scalability [13]. The axis of the maturity model combines the axis of the service component (core features to structure software) and the maturity levels (Figure 2) [32]:

- Ad Hoc Level: dedicated database and schema. Each customer has access to a different application through Web interfaces. With respect to the Service-level agreement (SLA), it depends on separated contracts reflecting the requirements for each user.
- Standardization Level: attempts to provide shared service. It shares the database using dedicated schema with configurable single tenants, allowing users to build their service model.

- Integration Level: multi-tenant environment with shared database and schema. There is a predefined shared instance and configurable options for each tenant. The business layer focuses on measurable SLA adaptations.

- Virtualization Level: focus on scalability and availability with the possibility to include load-balancing. User requirements are handled in the business process (instead of code customization) and optimized SLA policies per customer.

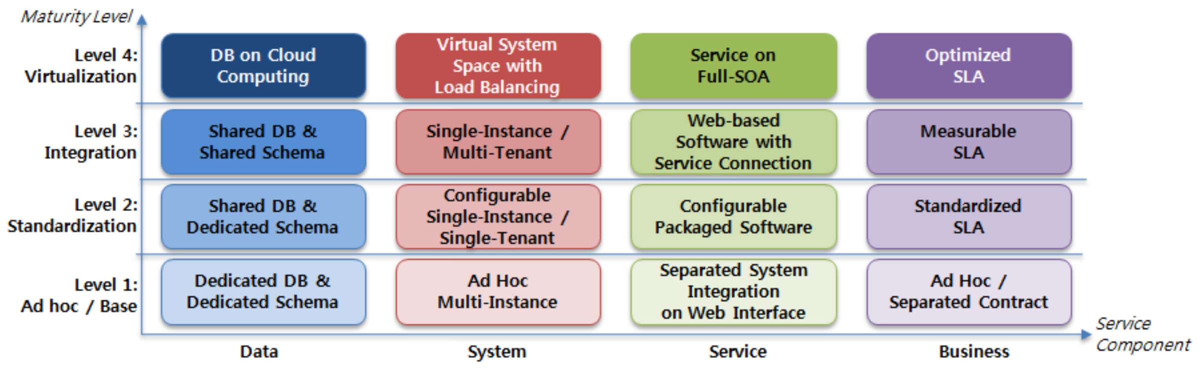

Maturity Level
Level 4: Virtualization
Level 3: Integration
Level 2: Standardization
Level 1: Ad hoc / Base
DB on Cloud Computing
Virtual System Space with Load Balancing
Service on Full-SOA
Optimized SLA
Shared DB & Shared Schema
Single-Instance / Multi-Tenant
Web-based Software with Service Connection
Measurable SLA
Shared DB & Dedicated Schema
Configurable Single-Instance / Single-Tenant
Configurable Packaged Software
Standardized SLA
Dedicated DB & Dedicated Schema
Ad Hoc Multi-Instance
Separated System Integration on Web Interface
Ad Hoc / Separated Contract
Data
System
Service
Business
Service Component

Figure 2: SaaS Maturity Model [13].

The different abstractions generated using these layers allows a PaaS provider to externalize the infrastructure to an IaaS provider and focus on their platform [32].

Some service providers are focused on reducing overall costs, while others are more expensive but focused on higher reliability and security. There clouds can be classified in several types [32]:

- Public: offering resources to the public.

- Private (also called internal): exclusive for a given organization. They offer more control at a higher cost than the public ones.

- Hybrid: some Cloud components are public, and others are private. The separation must be designed in advance.

- Virtual private clouds (VPC): a virtualization layer over public clouds. It virtualizes servers, applications and the network.

2 Security

SaaS models encourage businesses to shift their data management to another company. Security must be considered to keep customer trust. As a rule of thumb, multiple defence levels against any potential threats increases security.

SaaS applications allows to shift security management to the back end. This is a relevant change because one of the main security issues in computer sciences is end users. Splitting the security by design in multiple servers prevent a single point of failure and makes it more difficult for users to grant too many permissions (CORS policy is important here). Defence in the front end is still a nice to have, though. In desktop applications the code could be potentially obtained by reverse-engineering and modified to meet certain goals. Security in the front end is nice to have, but we must consider that any front end security can be bypassed.

Security must be considered by design, not to rely on security through obscurity. There are several security patterns (agnostic to the database management system) available for each data isolation approach [3]. In the scenario of multi-tenancy, these security approaches should be taken into account to provide security at different levels:

- Filtering: the communication between each tenant and the data source is restricted to an intermediary layer, allowing access only to their tenant data by design. It is possible to implement this by design in the code, using database access objects with automatic tenant identifiers filtering. Filtering should not depend on the ability of the programmer. The tenant id could be required for each request to the back end. For additional security, this tenant id can be stored in a temporal token previously generated and encrypted in the back end.

- Permissions: use access control lists (ACLs) to determine access data. Another desirable layer of security are Virtual Private Networks with Two-Factor Authentication.

- Encryption: cipher critical data in the database will prevent unauthorized individuals to extract information about the tenants even in case of data leaks.

Some security patterns are applicable in all the scenarios (Separate Databases, Shared Database with Separate Schemas and Shared Database with Shared Schema) [3]:

- Trusted Database Connections

- Secure Database Tables

- Tenant Data Encryption

Each scenario will be described below.

2.1 Trusted Database Connections

Impersonation is one method to improve security accessing databases information (Figure 3). In this approach, individual database users are granted with granular permissions at different levels: procedures, queries, views and tables. Any call to the database use the user's security context. It is possible to create a database user for each tenant, restrict its access only to strictly required stuff and modify the data layer of the code to use different database tenant accounts when pooling connections to the database.

Another approach is the trusted subsystem access method (Figure 4). In this scenario, database connections always use the application identity, without extra distinction for different tenants. It is still possible to increase security at code level using an extra permission layer (i.e. encrypted tokens generated in the back end), though. It is easier to handle but it would not allow deeper permission granularity at the database level, though.

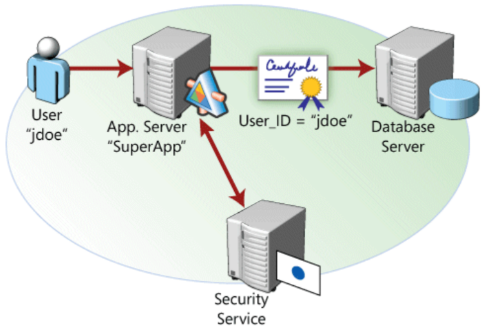


Figure 3: Impersonation database access [3].

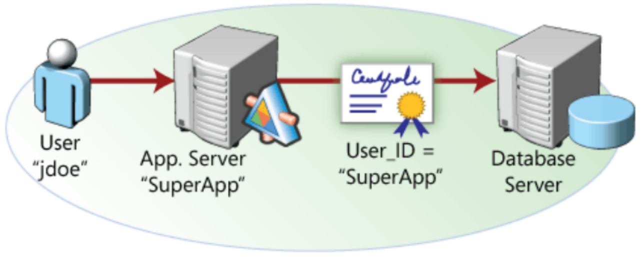


Figure 4: Trusted subsystem database access [3].

2.2 Secure Database Tables

It is possible to secure a database with table granularity using the SQL GRANT command. The following command will add access to a database user using the access control list for a given table. It grants restricted access only to a specific tenant user account:

```
GRANT SELECT, UPDATE, INSERT, DELETE ON [Table] FOR [User]
```

It is possible to restrict tenant access using database views. A view is a virtual table showing the results of a query [5], selecting rows from one or more tables. These views are not stored physically in the database. If there is a table called *Table* and several tenants {Tenant1, ..., TenantN} it is possible to prevent direct read access to *Table* and create several views

$$\{Tenant1TableView, \ldots, TenantNTableView\}.$$

Each TenantXTableView will be the result of a SELECT query filtering with the tenant id column.

```
CREATE VIEW Tenant1TableView AS
    SELECT * FROM Table WHERE tenant_id = get_current_tenant_id()
```

where *get_current_tenant_id()* is a procedure returning the current tenant identifier. A possible approach to define that procedure would be to create an intermediate table linking tenant ids with database users (using the security identifier SID of the database connection). It is possible to avoid the extra table directly hard coding the tenant id in the filter, but it would be less secure. It is relevant to highlight that only read access could be restricted using this approach because the tenants still need to insert and update rows in the original table.

Malicious/mistaken UPDATE, INSERT, DELETE statements could be prevented using a similar methodology with triggers (shown in Figure 5)

```
DELIMITER $$

CREATE TRIGGER
    TENANT_PREVENT_INSERT
     BEFORE INSERT
     ON Table FOR EACH
    ROW
BEGIN
  IF NEW.TenantID !=
    SUSER_SID() THEN
     -- Throw error
   END IF;
END$$

DELIMITER ;
```

```
DELIMITER $$

CREATE TRIGGER
    TENANT_PREVENT_UPDATE
     BEFORE UPDATE
     ON Table FOR EACH
    ROW
BEGIN
  IF OLD.TenantID !=
    SUSER_SID() THEN
     -- Throw error
   END IF;
END$$

DELIMITER ;
```

```
DELIMITER $$

CREATE TRIGGER
    TENANT_PREVENT_DELETE
     BEFORE DELETE
     ON Table FOR EACH
    ROW
BEGIN
  IF OLD.TenantID !=
    SUSER_SID() THEN
     -- Throw error
   END IF;
END$$

DELIMITER ;
```

Figure 5: MySQL base trigger examples to prevent insert/update/delete different tenant data.

2.3 Tenant Data Encryption

Cryptography can be used as a security layer and there are two main approaches:

- Symmetric: only one key is created, using it for encryption and decryption of data [3].
- Asymmetric (public-key cryptography): Two linked keys are generated, the public key will encrypt, and the private key will decrypt the data.

Asymmetric cryptography is slower than symmetric cryptography and could not fit in applications where performance is considered. In multi-tenancy is possible to add extra encryption security by creating different keys, one per tenant. It is more difficult to manage, though.

3 Extensibility

Extensibility measures effort required to extend a system's behaviour. Extensions of system's functionality increase the complexity and could make software maintainability harder. It is desirable to use one of the SOLID principles: Open-closed. Open-closed principle states that a system should be open for extension but closed for modification, preventing possible regressions in legacy code. An extensible design provides a framework to allow changes [14], small increments will improve software reusability.

Traditional software development requirements will be needed for appropriate extensibility [14]:

- High cohesion: degree to which part of the code forms a logically single and atomic unit. Example: classes that contain strongly related functionalities help to keep related parts of a code base in a single place.
- Low coupling: degree of interdependence between software modules. Changes in more independent modules will not heavily impact other modules.
- Interface-implementation separation allows you to change the implementation independently of the interface. Polymorphism allows several implementations of the same interface to do similar things in different ways.
- Dependencies: external dependencies are often needed to perform specific tasks. When a system grows is usually divided into several components, there will be internal inter-dependencies. It is possible to automatically generate the dependency graph in order to inspect coupling or dangling dependencies. External dependencies tracking allows for the recognition of possible migrations to avoid non-maintained or vulnerable dependencies. Extensibility would lead to fewer and cleaner dependencies and well-defined interfaces [14].
- Procedures to perform continuous integration: increase the pace and safety of code integration from several contributors in a system. Usage of automated tools for build, test and code analysis will allow faster feedback on new code's correctness [20]. Continuous Integration (CI) helps to achieve agile approaches for software development processes. The next step is Continuous Delivery, ensuring that software is always ready to be delivered. Continuous deployment will be the last step of this pipeline. In this phase, the software artifact(s) will be automatically launched and distributed to end-users when tests pass [20].

The extensible design fits well with Agile methodologies and iterative development, allowing dynamic prioritization [14] by looking at software as a growing entity.

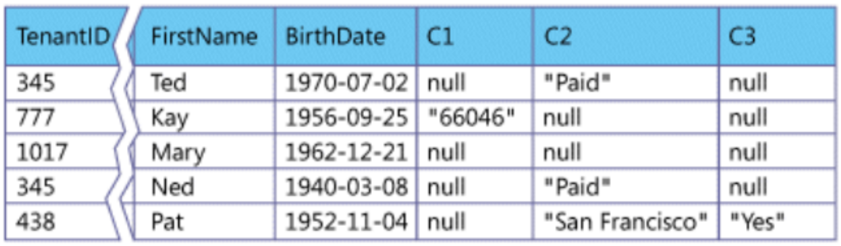
3.1 Multi-tenancy customization

It is possible to conditionally extend the functionality of a tenant by adding new features or modifying existing ones. There will be a set of tables/columns at the core of the application, but customized customer cannot always be addressed with a rigid data model.

There are several ways to extend the data model to handle custom functionality [3]:

- Preallocated Fields
- Name-Value Pairs
- Custom columns
- Columns with JSON type
- NoSQL databases

The data model can be customized creating personalized columns in certain table(s) to grant tenants extended functionality [3] (Figure 6). This black box approach allows determining how to use these fields depending on each customer, hiding actual data model complexity. Nullable strings could be used as a flexible data type for the custom fields, avoiding unnecessarily data type restrictions (castings can be done in the code).



TenantID	FirstName	BirthDate	C1	C2	C3
345	Ted	1970-07-02	null	"Paid"	null
777	Kay	1956-09-25	"66046"	null	null
1017	Mary	1962-12-21	null	null	null
345	Ned	1940-03-08	null	"Paid"	null
438	Pat	1952-11-04	null	"San Francisco"	"Yes"

Figure 6: Database table with preallocated fields C1, C2 and C3 [3]

Too few custom fields restrict tenants' customizability. Too many custom fields will lead to a sparse database with many unused fields. Name-Value pairs allow creating the desired customizable fields for each tenant [3] using one to many relationships with an external table (Figure 7). The number of pairs is limited by the maximum length allowed in the database for the name column. Actually this is not a relevant restriction because a tenant will not have millions of custom fields. When a record from *original_table* is fetched, a lookup in the table *name_value_table* will select all name-value pairs corresponding to the id of the external reference (lazy or eager loading can be used, depending on the specific use case). It is a good approach when specific customers require considerable flexibility, and the shared database approach is used. The disadvantages of name-value approach are:

- Black box approach with hidden types and possible database relationships
- Possible anti-pattern in the code:

```
if isTenant1():
    ...
elif isTenant2():
    ...
```

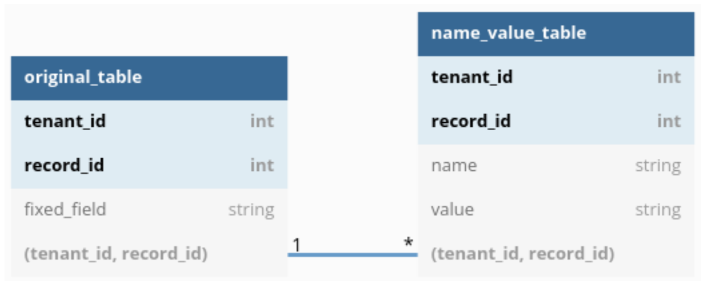
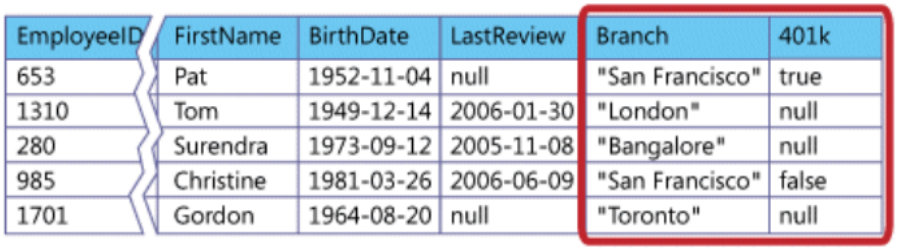


Figure 7: Tenants custom fields using name-value pairs [3]

Another extensible data model pattern is adding tailored columns to tenants' tables (Figure 8). The tailored database columns approach can be considered more appropriate in separate database/schema scenarios. There would be extra unneeded fields for several customers in a shared database environment when different tenants have radically different requirements. With a shared database, the database complexity would be the sum of all the tenants' complexities because it would include all the tailored columns. It should be possible to add these features dynamically to the clients, and the code must work regardless of the combination of the features (if there are no incompatibilities). Therefore the inner code complexity is similar with all the strategies used to separate tenants' data.



EmployeeID	FirstName	BirthDate	LastReview	Branch	401k
653	Pat	1952-11-04	null	"San Francisco"	true
1310	Tom	1949-12-14	2006-01-30	"London"	null
280	Surendra	1973-09-12	2005-11-08	"Bangalore"	null
985	Christine	1981-03-26	2006-06-09	"San Francisco"	false
1701	Gordon	1964-08-20	null	"Toronto"	null

Figure 8: Custom database columns extensibility approach [3]

From a performance point of view, a customer with simple requirements might not need the database overhead of extra database columns. With entire custom database tables, the possible overhead can be reduced at the code level using lazy loading and preventing N+1 issues (unnecessary database calls while iterating over a collection of entities, fetching external reference entities in a loop instead of a single fetch). The overhead is not a big deal in practice.

Databases can store JSON values using either strings (needs to be manually deserialized after fetch) or with native JSON types for database column definition. The JSON format allows storing dynamic fields using a tree structure. Some databases, like PostgreSQL, allow storing JSON in a native format, offering custom queries to filter data using the key/values of the JSON cells. That native approach is better than directly storing the JSON as a string. It also provides flexibility inside data structured in tables.

Another possibility to customize client requirements is to use NoSQL databases. The relational model in SQL databases allows for more structured data since the database contains foreign keys [2]. On the other hand, NoSQL databases do not have these foreign keys, allowing more flexibility while storing data [2]. This flexibility relegates the responsibility of checking the possible relationships between the data manually in the code, but it is a good fit for non-structured data. In other section, the performance differences of SQL versus NoSQL databases will be covered.

Any custom field will lead to changes in the business logic, the presentation logic, or both, regardless of the extensibility pattern(s) used. Any change to the business logic increases the entropy of the system and increases the risks of including software regressions.

3.2 Software regressions

A software regression is a special type of bug where a certain feature was working before and stops working. Regressions can be produced due to changes in the code or the environment (i.e. system upgrades, environment variables, ...). Regressions can be classified in:

- Functional: a feature that is no longer working.
- Performance: a feature is still working but more slowly or uses more resources (i.e. memory) than before.

Testing can reduce regressions likelihood, and tests are either manual (sometimes involving Quality Assurance positions) or automatic.

3.2.1 Automatic testing

Automatic tests will catch bugs before production release. Several strategies that can be used:

- End to end tests (E2E): involves entire workflows, typically front end, back end and database(s).
- Integration tests: specific use cases involving several system components.
- Unit tests: test cases involving only one component, mocking other components. For example: in the back end, the database calls are fake (just mocked), making the tests much faster.

A convenient relationship between the number of tests for each strategy is shown in the test pyramid:

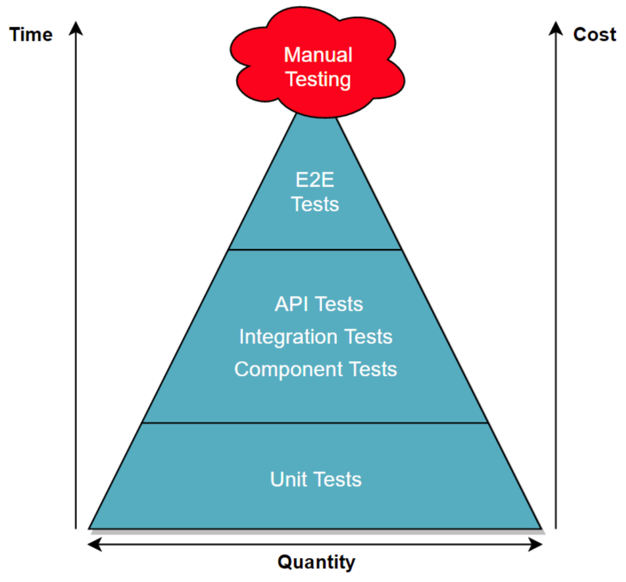


Figure 9: Test pyramid [9].

The inverted testing pyramid (fewer unit and integration tests, with an emphasis on automated and manual functional testing) is considered an anti-pattern, reducing the responsiveness, maintainability, and reliability of the test setup [9]. Unit tests help identify faults in earlier phases [9] and must be the backbone of any software product.

Regression testing validates that new changes applied to a program do not break existing logic [19]. Unit tests can be introduced in the continuous integration loop because their feedback is almost instantly. End to end and integration tests do not scale well. so it is not always possible to include all of them in the continuous integration loop. Regression test selection techniques reduce the cost in time by selecting only the tests related to a modified program (i.e. two consecutive commits in a version control system) [21].

Other types of tests should also be considered [30]:

- Performance tests: check the system when it is under significant load, they are quite costly to implement and run.
- Smoke tests: check basic functionality of the application.
- Security Testing: check how the system reacts to internal and/or external threats (i.e. penetration testing).
- Mutation tests: it is a formal way to measure the quality of existing tests, weak tests can produce a false sense of safety. The code of a program is slightly modified, which is called a

mutant, and then the tests are run. If the tests are robust enough, at least one should fail and the program modification is called a killed mutant. It is measured as

$$Mutation\ score := \frac{Killed\ mutants}{Total\ number\ of\ mutants} \cdot 100.$$

This approach inherits the scalability of the tests, and it is more suitable to verify unit tests.

The code of a program can be described as a graph [21]. Two different commits using a version control system are two different programs, i.e. they are represented as two graphs. A rest API call (or any user action that only requires code execution in the front end) towards this program can be represented as a path inside this graph. An integration test or end-to-end test can be represented as an ordered set of paths within one graph. A pull request (updated with respect to the pull request destination branch) represent a different program. Therefore it is possible to extract a subset of the integration/end-to-end tests in order to detect regressions earlier and with lower costs. This approach is more clever than traditional ones to handle slow tests, like smoke tests (although they are compatible approaches), because it is possible to run only the tests whose behaviour will change after merging.

Software programs tend to be more complex over time. The number of tests won't (or should not) be reduced, but systematic approaches to handle this complexity are definitely nice to have. Sometimes it will still be infeasible to run all the tests (ex. changes in one core part of the program) before merge a pull request. In those scenarios it is possible to select a subset of tests maximizing the edges of the graph covered. At a certain level of development the execution time of slow tests will force teams to schedule its execution daily, weekly, etc. This approach allows to statically analyze the possible root cause(s) when tests fail, searching the commits that modified the parts of the graph involved in these tests.

4 Concurrent systems modeling with Petri Nets

In pure functional programming paradigms, a program is a composition of pure functions. A pure function verifies two properties:

- For the same input value(s), the output is always the same.
- There are no side effects, no variable/state outside of the scope of the function mutated after the function application.

A pure function can be considered equivalent to a mathematical function. This approach allows certain optimizations like lazy evaluation or memoization. On the other hand, functional programming cannot fully use pseudo-random numbers (since they usually use the execution time as a seed for randomization) or program state. Program state is a source of potential failures and should be avoided when possible, but most the programs depend on databases continuously changing their state.

Most of the content of this section is inspired in the lecture notes of the subject 'Concurrent programming' provided by Manuel Isidoro Capel Tuñón in [29]. A reactive system interact with the environment, it cannot be represented as a composition of pure functions. Concurrent systems are a subset of reactive systems, there are sequential reactive systems that are not concurrent.

Definition 3. **Fair Transition System (FTS).** It is a quadruple (Σ, T, Σ_0, R) where

- Σ is a set of states. The state represents the value of the system variables at a precise moment.
- T are the transitions of the system. A transition, $t \in T$, is a function

  $$t : \Sigma \to \mathscr{P}(\Sigma) \qquad (\mathscr{P}(\Sigma) \text{ represents all the possible subsets of } \Sigma),$$

  from a state $s \in \Sigma$ to a subset of Σ, $t(s)$. The subset $t(s)$ is called the successors of s.
- Σ_0 are the possible initial states of the system, $\Sigma_0 \subseteq \Sigma$.
- R are the correctness requirements of the system. They limit the frequency of transitions.

A Fair Transition System can represent any king of concurrent system, based on shared memory or message passing [29]. In FTSs is possible to define an execution:

Definition 4. **FTS execution.** A sequence of states connected by transitions:

$$\sigma : s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \dots \qquad s_i \in \Sigma, t_i \in T.$$

An execution represents the evolution of the global state of a system after several transitions.

4.1 Petri Nets

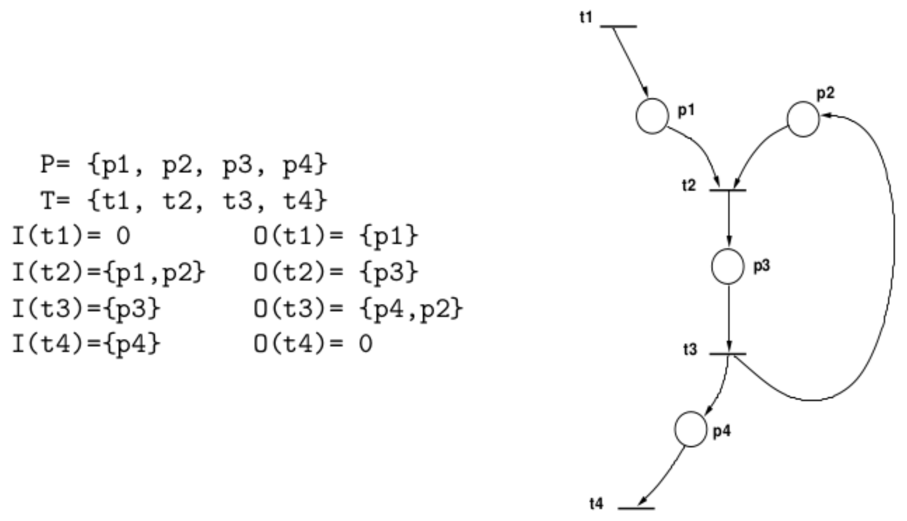
Studying the structure of Fair Transition Systems and its possible executions will provide relevant properties about the system's correctness. The first approach to study these systems are Petri Nets.

Definition 5. **Petri Net** is a directed multigraph (a graph with two kinds of nodes) with annotations [29]. It is defined as a 4-tuple $(P,\ T,\ I,\ O)$ where

- $P := \{p_1,\ \dots\ ,\ p_n\}$ are the places (first kind of node), they represent the state variables of a discrete event system.
- $T := \{t_1,\ \dots\ ,\ t_m\}$ are the transitions (second kind of node) transforming the values of the places.

  The arcs of a Petri Net connect places with transitions (and vice versa). Connections between two places or two transitions are not allowed. These arcs are defined using the functions $I, O : T -> Bag(P)$. $Bag(P)$ is the multiset of P, representing all the possible subsets of elements of P. This definition allows having several arcs from one place to one transition since the multi-set allows several identical items. The values returned by the functions I, O depend on the specific Petri Net.
- $I(t) : T -> Bag(P)$ input function (or preset), it is the set of input places for a transition.
- $O(t) : T -> Bag(P)$ output function (or post-set), it is the set of output places for a transition.

  A transition $t \in T$ is considered a fork when $|O(t)| > 1$, when $|I(t)| > 1$ it is considered a join. Respectively, for places, a place $p \in P$ is considered a collector when $|\{t : p \in O(t)\}| > 1$, when $|\{t : p \in I(t)\}| > 1$ it is considered a distributor.



```
  P= {p1, p2, p3, p4}
  T= {t1, t2, t3, t4}
I(t1)= 0          O(t1)= {p1}
I(t2)={p1,p2}     O(t2)= {p3}
I(t3)={p3}        O(t3)= {p4,p2}
I(t4)={p4}        O(t4)= 0
```

Figure 10: Graphic representation of a Petri Net.

Petri Nets can describe dynamic systems using discrete events. It is possible to add concurrency restrictions in the interactions between these asynchronous events. The mathematical nature of Petri Nets allows us to analyze the systems using different king of Petri Nets: P/T, High-Level PN, Colored Petri Nets, Timed Petri Nets, etc.

Definition 6. **Pure Petri Net**, it is a Petri Net where there is no place simultaneously being the input and the output of a transition.

Definition 7. **Regular Petri Net**, it is a Petri Nets with a maximum of one arc per direction between a place and a transition.

One arc starting at place p_i to transition t_j and other arc from t_j to p_j is a valid combination in regular Petri Nets. The Figure 10 represents a pure regular Petri Net.

4.2 Marked Petri Nets

To study the executions in Petri Nets, we need to define the marks stored in their places. Intuitively, these marks will describe the state of the net at a given time. The mark distribution represents the number of marks for each place of the Petri Net at a given moment. A marking M is represented as a vector, $M = (M_1, ..., M_{|P|})$, where each element $M_i \in \mathbb{N}_0$ represents the number of marks at place p_i. Formally, the marking function is

$$M : P \to \{0,\ 1,\ \dots\ ,\ |P|\}$$

$$M(p_i) := M_i,$$

and it represents the current global state of a system. This definition allows us to define Marked Petri Nets as follows:

Definition 8. **Marked Petri Net** is a special kind of Petri Net with an initial marking. It is formally described as a 5-tuple $R := (P,\ T,\ I,\ O,\ M_0)$ where M_0 is the initial marking.

In some specific nets, a place has its marking dependent on the marking of other places. They are called implicit places and, under certain circumstances, can be removed from the net without changing its behaviour [29].

An example of a graphical representation of a Marked Petri Net is shown in Figure 11.

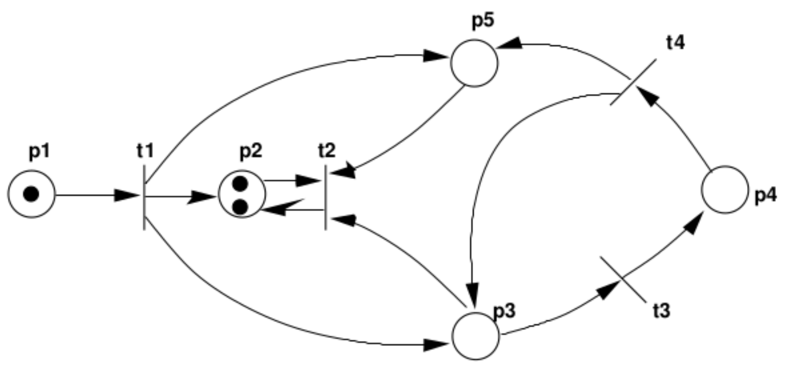


Figure 11: Marked Petri Net with one mark at place p_1 and two marks at place p_2 [29].

Now, some definitions are required to describe the executions in Marked Petri Nets. The multiplicity,

$$n : P \times T \to \mathbb{N}_0$$

$$n(p,t) := |[x \in I(t) : x = p]|$$

is the number of arcs from one place to a transition. It is possible to create an equivalent definition to define the arcs' multiplicity from a transition to a place. There is an essential property of transitions related to multiplicity:

$$t \in T \text{ is enabled} \qquad \iff \qquad \forall p \in I(t) \quad M(p) \geq n(p,t).$$

Definition 9. **Reachability set of a Marked Petri Net.** Given a Marked Petri Net $R = (P,\ T,\ I,\ O,\ M_0)$, the reachability set $\mathscr{M}(R, M_0)$ is the set of all the reachable markings starting from M_0.

A transition can be fired if and only if it is enabled. The marking M of the net will change following this function:

$$\delta(M,t) : \mathscr{M}(R,\ M_0) \times T \ \to\ \mathscr{M}(R,\ M_0)$$

$$\delta(M,t) := \begin{cases} M(p) + n(p,t) & \text{if } p \in O(t) \wedge p \notin I(t) \\ M(p) - n(p,t) & \text{if } p \notin O(t) \wedge p \in I(t) \end{cases}.$$

Intuitively, we can think of transition firing as a way to get one mark from each input place and transform them to one mark for each output place. Clearly, the total number of marks in the net changes when a transition t is fired and verifies $|I(t)| \neq |O(t)|$. We can use Figure 12 to see an example of a transition firing. The transition t_1 is enabled because all its input places ($\{p_1\}$) have a number of marks (one) greater or equal than the number of arcs from p_1 to t_1 (one). After the firing, since $n(p_1,t_1) = 1$, place p_1 will have $1 - n(p_1,t_1) = 1 - 1 = 0$ marks, and place $p_2 \in O(t_1)$.

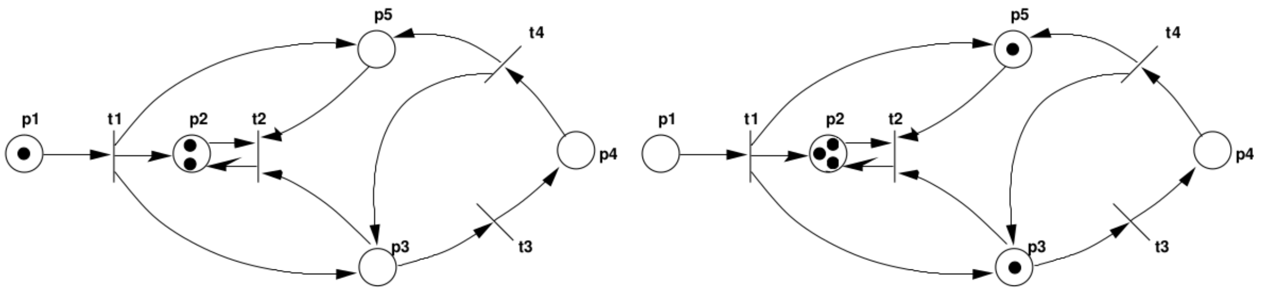


Figure 12: Evolution of a Marked Petri Net after firing of transition t_1.

Two enabled transitions conflict when the act of firing one prevents the firing in the other. If two transitions are enabled, both can be fired, but not simultaneously. The flexibility provided with the properties of non-determinism and no simultaneous firing of transitions allows concurrency (Figure 13) and conflicts (Figure 14) modeling.

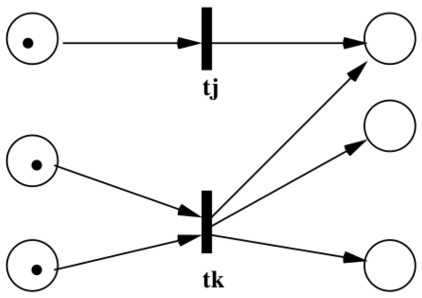


Figure 13: Concurrency modelling with two possible execution sequences, t_jt_k or t_kt_j [29].

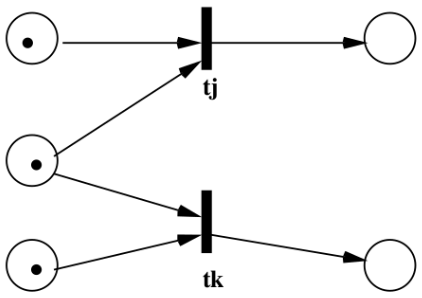


Figure 14: Conflict modelling, either t_j or t_k can be fired [29].

A marking M' is immediately reachable from an initial marking M if and only if an enabled transition can be fired resulting in M'. M' is reachable from M if there is a sequence of fired transitions from M resulting in M'. Of course, if a marking is immediately reachable, it is also reachable.

4.3 Petri Nets properties

Given a Marked Petri Net $R = (P,\ T,\ I,\ O,\ M_0)$, then one property of its places is k-limitation. A place is k-limited for an initial marking M_0 when

$$\exists k \in \mathbb{N}\ :\ M'(p_i) \leq k \quad \forall M' \in \mathscr{M}(R, M_0).$$

This property can be extended to the whole net.

Definition 10. **K-limited Marked Petri Net.** A Net is called k-limited for the initial marking M_0 if and only if all its places are k-limited for that initial marking.

If the Net is k-limited for any $k \in \mathbb{N}$, then the reachability set is finite.

Definition 11. **Secure Net.** A 1-limited Net.

Definition 12. **Structurally limited Marked Petri Net.** It is a special kind of Net limited for all the possible initial finite marking.

Another interesting property is liveness. liveness will be defined for transitions and then generalized to the entire Net.

Definition 13. **Alive transition.** A transition $t \in T$ is alive when for all the markings in the reachability set of the Net, there is a sequence of firings that will enable the transition t. Formally,

$$\forall M \in \mathscr{M}(R, M_0) \quad \exists \sigma : M \to M' \quad \text{such that} \quad t \in \sigma.$$

liveness ensures that no transition firing sequence will block a transition, preventing it to be potentially enabled again. This property can be generalized:

Definition 14. **Alive Marked Petri Net** for the initial marking M_0 when all the transitions are alive for that initial marking.

Liveness can be relaxed to include more Nets when an initial marking to make the Net alive exists.

Definition 15. **Structurally alive Marked Petri Net** is a Net allowing an initial finite marking M_0 where the Net is alive.

The alive property is more general. The relationship between these two liveness properties is shown in Figure 15.

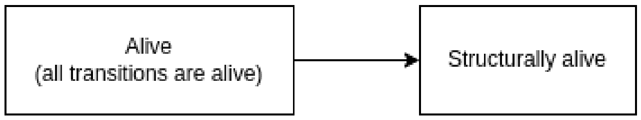


Figure 15: Liveness properties relationship for Marked Petri Nets [29].

A Marked Petri Net is **cyclic** for the initial marking M_0 when there is a transition firing sequence reaching again M_0 for any reachable marking:

$$\forall M \in \mathscr{M}(R, M_0) \quad \exists \sigma : M \to M_0.$$

From the concurrency point of view, it is critical to categorize the conflicts to find better ways to represent them. There are two kinds of conflicts, structural and effective.

Definition 16. **Structural conflict (Figure 16).** It occurs when a place $p \in P$ is linked with more than one output transition, but these output transitions can not be simultaneously enabled for any reachable marking.

Definition 17. **Effective conflict between two transitions** $t_1, t_2 \in T$ **(Figure 17)** for an initial marking M_0 if there is a potential marking $M \in \mathscr{M}(R, M_0)$ that will enable both transitions, and, after firing one of them, the other will lose its enabled status.

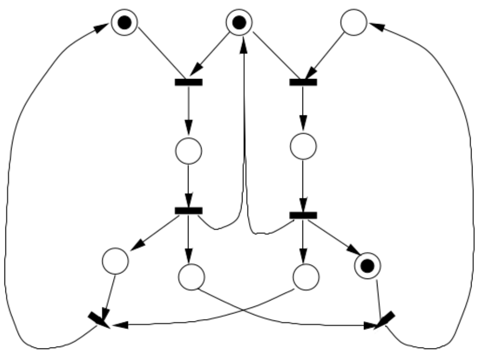


Figure 16: Structural conflict, it is not possible to have the three upper places simultaneously marked [29].

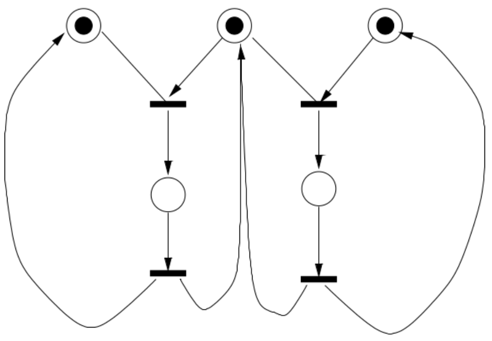


Figure 17: Effective conflict between the two upper transitions [29].

Both definitions rely on the initial marking of the Net since they depend on the reachable markings. As a consequence of the definitions, the intersection of the structural conflicts and effective conflicts is empty. It is possible to model the resources of a system using the markings. For these scenarios, it is relevant to distinguish Nets where resources are created/destroyed/kept.

Definition 18. **Strictly conservative Marked Petri Net**, a given Net $R = (P,\ T,\ I,\ O,\ M_0)$ verifying that the total amount of marks is constant

$$\forall M' \in \mathscr{M}(R, M_0), \quad \sum_{p_i \in P} M'(p_i) = \sum_{p_i \in P} M(p_i).$$

From the strictly conservative property follows that no transition can reduce or increment the number of marks on the Net. That behaviour would produce a marking with a different amount of marks than the initial one. As a consequence, for all the potentially enabled transitions $t_j \in T$ the property $|I(t_j)| = |O(t_j)|$ is verified. There is also a relaxed conservative property:

Definition 19. **Conservative Marked Petri Net**, a given Net R where, for specific weights in the places, $w = (w_1, ..., w_n)$, the weighted sum of the markings is constant:

$$\forall M' \in \mathscr{M}(R, M_0), \quad \sum_{p_i \in P} w_i \cdot M(p_i) = \sum_{p_i \in P} w_i \cdot M('p_i).$$

Definition 20. **Repetitive MPN.** It is a Marked Petri Net containing a transition sequence including all the transitions. That sequence must finish in the same initial marking, like:

$$M \xrightarrow{\sigma} M \qquad \wedge \qquad \forall t \in T, \quad t \in \sigma \qquad M \in \mathscr{M}(R, M_0).$$

Finally, the security properties summary appears in Figure 18.

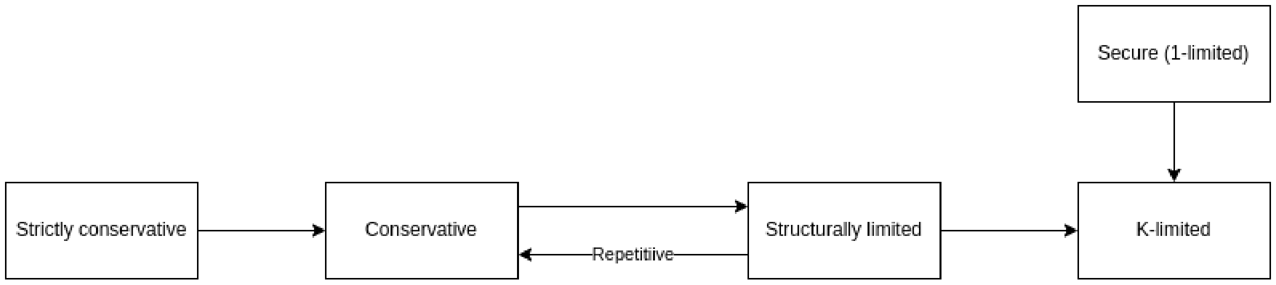


Figure 18: Security properties relationship for Marked Petri Nets [29]. The implication from structurally limited to conservative when the net is repetitive is not proven.

The next section presents an algorithmic approach for studying the possible markings of a net.

4.4 Reachability tree generation

Most of the described properties depend on the reachability tree. The reachability problem consists of, given a specific marking in a Petri Net, verifying if the marking belongs to $\mathscr{M}(R, M_0)$ or not. Each node is a marking on the Net, represented as a vector whose i-component is the number of marks for the place p_i on the Net, i.e.

$$(M(p_1),\ M(p_2),\ \dots\ ,\ M(p_n)) \qquad p_1,\ p_2,\ \dots,\ p_n \in P.$$

It is possible to define a partial order relationship between two markings $M_1 = (a_1, ..., a_n)$, $M_2 = (b_1, ..., b_n)$ such that

$$\begin{aligned} M_1 \leq M_2 \quad &\Longleftrightarrow \quad \forall i \in \{1, ..., n\} : a_i \leq b_i, \\ M_1 < M_2 \quad &\Longleftrightarrow \quad M_1 \leq M_2 \wedge \exists i \in \{1, ..., n\} : a_i < b_i. \end{aligned} \tag{1}$$

The algorithm to compute the finite reachability tree uses a symbol ω to represent the infinite, and there is no change in that value if there is a sum or subtraction.

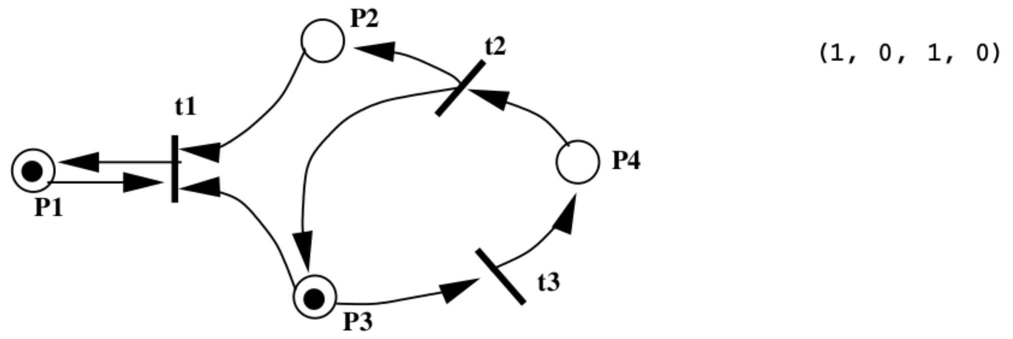
Algorithm 1. Reachability tree generation

- Starts from the initial marking M_0.
- Each enabled transition will generate a node (unless that node is already in the tree).
- After the creation of the next node M_j, if there is another node in the tree M_i, such that,

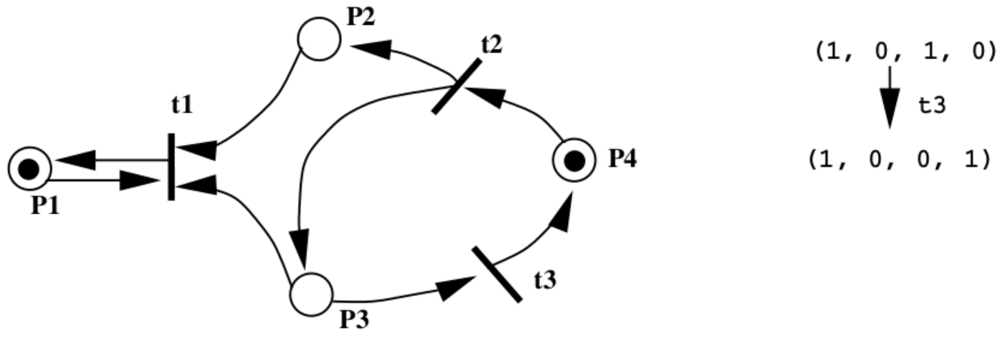
$$M_j \in \mathscr{M}(R, M_i) \quad \wedge \quad M_j \geq M_i,$$

  then all the places where M_j has more marks than M_i are marked with ω.

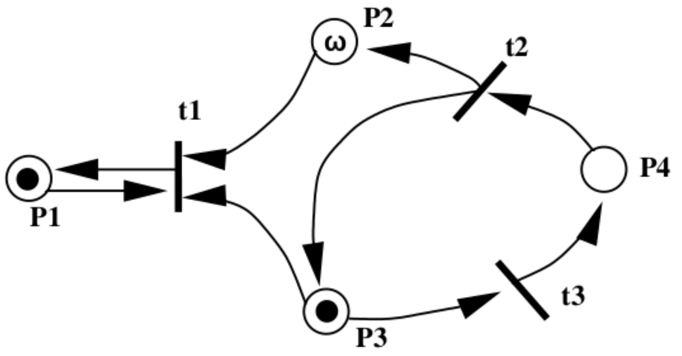
Example of the computation of a reachability tree:



The first step of the tree represents the initial marking $(M(p_1),\, M(p_2),\, M(p_3),\, M(p_4))$.



After firing of the only enabled transition (t_3), it shifts the mark p_3 to p_4.

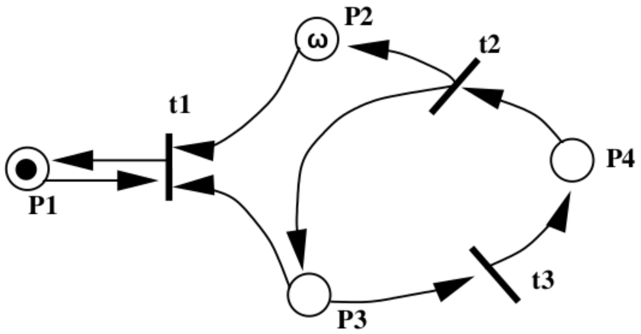


$$\begin{tikzcd}(1, 0, 1, 0) \arrow[d, "t3"] \\ (1, 0, 0, 1) \arrow[d, "t2"] \\ (1, \omega, 1, 0)\end{tikzcd}$$

Firing of t_2 removes mark from p_4 and generate marks in p_2 and p_3. This Net is **not conservative** since the total number of marks is potentially infinite. There is no combination of weights keeping the weighted sum of marks constant. Notice that this marking is reachable from the initial marking and greater than it (using (1)). The set of places with more marks now than in the initial marking is $\{p_2\}$. Then, the number of marks in p_2 is replaced with ω. Without the ω rule, another sequence of transition firings (t_1, t_2) would produce another node. To avoid infinite nodes in the reachability tree, the infinite sequence of nodes

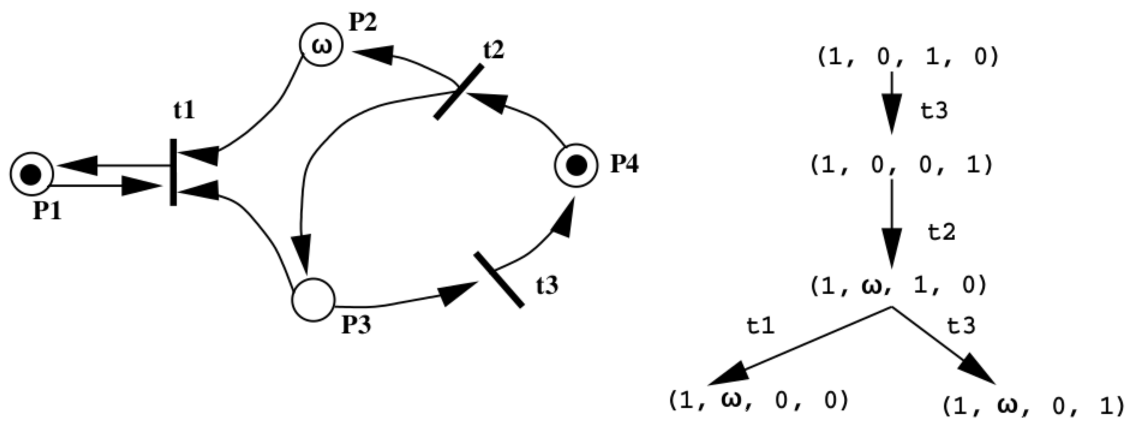
$$\{(1,1,1,0),\ (1,2,1,0),\ ...,\ (1,n,1,0),\ (1,n+1,1,0),\ ...\}$$

is simplified to $(1,\omega,1,0)$ allowing the algorithm to be finite for any Net. **When there is any ω in the reachability tree the Net is not k-limited** for any $k \in \mathbb{N}$, therefore is not a secure Net. Now, it is clear that a node does not strictly represent a marking, but a equivalence relation between comparable markings.



$$\begin{tikzcd}[column sep=small] & (1, 0, 1, 0) \arrow[d, "t3"] \\ & (1, 0, 0, 1) \arrow[d, "t2"] \\ & (1, \omega, 1, 0) \arrow[dl, "t1"'] \\ (1, \omega, 0, 0) & \end{tikzcd}$$

Now, all the inputs for transition t_1 have marks (indeed, p_2 has potentially infinite marks). Both, t_1 and t_3, are enabled now. After firing of t_1 the mark in p_3 disappears but p_2 still has ω marks. After this firing, t_1 is no longer an alive transition. Indeed, no transition is potentially enabled again, and **the liveness property is not verified**.



If instead of t_1, t_3 is fired then there is another node in the tree.

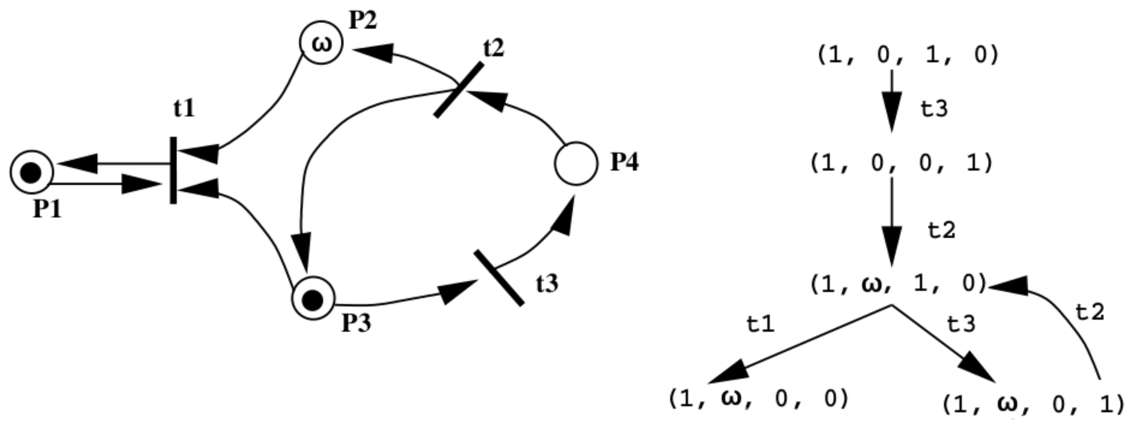


Figure 19: Final reachability tree generation.

The only missing enabled transition is t_2, after that, there is no node to explore and the final reachability tree is shown in Figure 19.

In the example above, the Net is not alive because a node in the tree did not have any available transition. As it turns out, if the reachability tree has any leaf node, then the Net is not alive. The tree can be adapted to create an algorithm to check the liveness property. The transition firings will now be nodes, and the marking relationships will be edges. In the transformed tree, all the nodes must be potentially reachable from any other node to verify the liveness property. As a side note, if ω does not appear in the reachability tree, it is possible to check limited and conservative properties using the generated tree [29].

It is the moment to introduce some static properties that do not depend on the initial or current marking of a net. They are called invariant properties. It is possible to prove some Petri Nets properties using matrix operations [29]. Using the number of arcs between a place and a transition, it is possible to define two matrices:

$$D^{-}[j,i] := n(p_i, I(t_j)), \qquad D^{+}[j,i] := n(p_i, O(t_j)) \qquad j \in \{1,...,|T|\},\ i \in \{1,...,|P|\}.$$

The dimension of the matrix is $|T| \times |P|$. It could not be a square matrix since the number of places could differ from the number of transitions.

Definition 21. **Incidence Matrix**, defined as

$$D := D^{-} - D^{+} \qquad D^{-}, D^{+} \in M_{|T|\times|P|}(\mathbb{N}^{+}_{\not 0}).$$

The only remarkable ambiguity is that zeros in D could mean either no link (pure nets) or an equal number of inputs and outputs. As a consequence, the same matrix D can represent different nets. The next step is to define mathematically some of the previous Petri Net concepts.

Definition 22. **Enabled transition**. Given a transition t_j and a vector $e_j = (0,...,1,...,0)_{1\times|T|}$ with all the elements zero except the jth, which is a one, then

$$t_j \text{ is enabled for the marking } M \quad \Longleftrightarrow \quad M \geq e_j \cdot D^{-}.$$

Definition 23. **Transition function**, given a marking M and a transition t_j, the generated marking of the net after t_j firing is:

$$\delta(M, t_j) := M - e_j \cdot D^{-} + e_j \cdot D^{+}.$$

The incidence matrix simplifies the study of the transitions of a Petri Net because it allows us to use only one matrix from now on. Following the definitions of the incidence matrix and the transition function, it is possible to deduce the next corollary without formal proof.

Corollary 1. $\delta(M, t_j) = M + e_j \cdot D$.

For a finite list of transitions $t_a, t_b, ... \in T$ it is possible to generalize the transition function using $\sigma = t_a, t_b, ...,$

$$\delta(M, \sigma) \;=\; M + (e_a + e_b + ...) \cdot D \;=\; M + f(\sigma) \cdot D.$$

$f(\sigma)$ represents the firing vector and its value at index i represents the number of firings of transition t_i. Now it is possible to define the state equation:

Definition 24. **Petri Net state equation** for a given initial marking M_0:

$$M' = M_0 + f(\sigma) \cdot D.$$

Having a formal description of the evolution of the markings of a net it is not possible to assume that any mark combination is possible. The notation for the possible firing sequences is defined in the next corollary.

Corollary 2. Not every firing sequence combination is valid. The valid sequences are

$$L(R,M_0) := \{\sigma \mid M_0 \xrightarrow{\sigma} M\} \subset T^{\mathbb{N}}.$$

Corollary 3. σ does not fix any order, different sequence orders could lead to the same $f(\sigma)$. That's why it is not possible to infer a sufficient vivacity condition with this notation.

The reachability problem is equivalent to the existence of a solution X verifying

$$M' = M + X \cdot D \qquad X \in L(R,M).$$

Proposition 1. Given a Petri Net M with incidence matrix D, then

$$\exists Y \in (\mathbb{N}^+)^{|T|} \; / \; D_{|T|\times|P|} \cdot Y^T_{|P|\times 1} = 0 \quad \Longleftrightarrow \quad M \text{ is conservative.}$$

Proof.
$\Longrightarrow$
From the state equation,

$$M' = M + X \cdot D \implies M' \cdot Y = M \cdot Y + X \cdot D \cdot Y \overset{(*)}{\implies} M' \cdot Y = M \cdot Y.$$

Where $(*)$ means the precondition ($D_{|T|\times|P|} \cdot Y^T_{|P|\times 1} = 0$). Y is composed of positive natural numbers, so one firing verifies the assertion. A reachable marking is a finite sequence of firings. As a result, the Petri Net M is conservative.

$\Longleftarrow$

$$\forall M' \in \mathscr{M}(R,M_0), \quad \sum_{p_i \in P} w_i \cdot M(p_i) = \sum_{p_i \in P} w_i \cdot M('p_i) \quad w_i > 0 \quad \forall i \in \{1,\ldots,w_{|P|}\},$$

then, $Y = (w_1,\ldots,w_{|P|}) \in (\mathbb{N}^+)^{|P|}$. Again, from the state equation.

$$M' = M + X \cdot D \implies M' \cdot Y = M \cdot Y + X \cdot D \cdot Y \overset{(*)}{\implies} 0 = X \cdot D \cdot Y.$$

Where $(*)$ comes from the conservative condition ($M' \cdot Y = M \cdot Y$). X can not be zero because it is a non-empty sum of elements e_i of the standard basis. In conclusion, $D \cdot Y = 0$. □

Definition 25. **Marking invariant**, property verified for all reachable markings from a starting one.

In conservative Petri Nets, the invariants do not depend on the initial or current marking and derive from the proven condition. These invariants are vectors Y_i from the system $D \cdot Y^T = 0$, having $|P|$

equations and $|T|$ unknown values. In general, there are no restrictions for the elements of each vector Y_i. Defining

$$r := range(D) \quad 0 \leq r \leq \min(|P|, |T|),$$

if $(|P| - r) > 0$ we have a linear span

$$B = \{Y_i \ / \ Y_i = (Y_{i,1}, ..., Y_{i,|P|}), \ i \in \{1, ..., |P| - r\}, \ \ Y_{i,j} \in \mathbb{Z}\}.$$

B is called the system of right overridjes for the incidence matrix D, and any solution for $D \cdot Y^T = 0$ is a linear combination of vectors from B. Since all the vectors from B must verify the invariant property, then this restriction is also required:

$$M \cdot B = M_0 \cdot B \quad M \in \mathbb{N}^{|P|}.$$

It is possible to verify that a marking is not reachable by checking all these conditions.

Definition 26. **Conservative component** of a Petri Net with incidence matrix D is each vector $Y \in \mathbb{N}^{|P|}$ such that $D \cdot Y^T = 0$.

Definition 27. **Support of a conservative component** Y is the set of places associated with non-zero elements of Y, denoted as $||Y||$.

The producer-consumer is a known concurrency problem. It models the data flow between two different processes, one producer and one consumer. Producer continuously creates data and sent to a common buffer, where the consumer can consume data at its own pace. There are many variants, with several producers/consumers/buffers. In its simpler version, one producer creates data and send it to one buffer, while only one consumer use the data in the buffer. Marked Petri Nets can model a network representing the producer-consumer problem, as shown in Figure 20. In the figure, there are three producers (not completely independent since they are connected to each other), represented as three subgraphs,

$$\{\{p_1, p_2\}, \ \{p_5, p_6\}, \ \{p_3, p_4\}\}.$$

There are two buffers ($\{t_2, t_3\}$), and the consumers are the same than the producers, the three subgraphs. The firing of fork transitions like t_2, t_3 produces two marks in the graph and removes just one. There is no way to reduce the number of marks since there is no joint transition. In conclusion, the net represented in Figure 20 is not secure because it is not 1-limited.

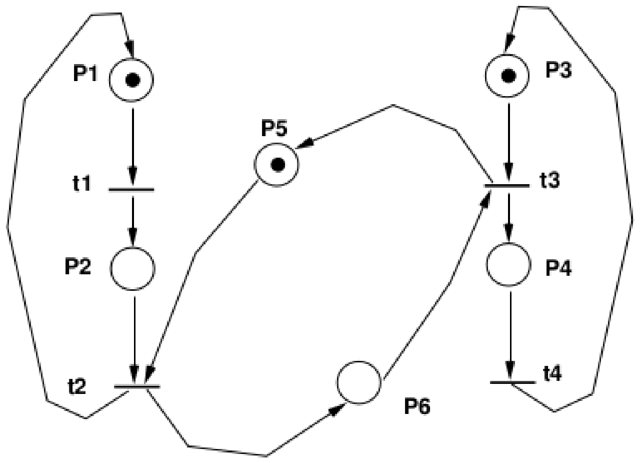


Figure 20: Producer-consumer with conservative components $\{\{p_1,\ p_2\},\{p_5,\ p_6\},\{p_3,\ p_4\}\}$

It is easy to identify that the conservative components from Figure 20 represent different substructures in the Petri Net, joined by transitions t_2, t_3. The computation of its conservative invariants derives from:

$$D \cdot Y^T = 0 \implies \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & -1 & 1 \\ 0 & 0 & -1 & 1 & 1 & -1 \\ 0 & 0 & 1 & -1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{pmatrix} = 0.$$

Definition 28. **Fundamental conservative component set** is the linear span $\mathscr{Y} := \{Y_1,\ \dots\ ,\ Y_n\}$ with fewer elements able to generate any conservative component.

$$Y = \sum_{Y_i \in \mathscr{Y}} k_i \cdot Y_i \qquad k_i \in \mathbb{N}_0,$$

and the elements Y_i are called elemental components.

This fundamental set is finite and unique [29]. Each elemental component represents a restriction in the evolution of the marks. A previous section presented the concepts of structural and effective conflicts. Now we provide a sufficient condition to verify that two places can not be enabled simultaneously.

Proposition 2.**Mutual exclusion between two places**. Given a conservative component Y, and two places in its support, $p_i, p_j \in \|Y\|$, then

$$Y(p_i) + Y(p_j) > M_0 \cdot Y \quad \implies \quad p_i, p_j \text{ can not be simultaneously enabled.}$$

Proof. Using the invariant relationship for a given marking $M \in \mathscr{M}(R, M_0)$:

$$M_0 \cdot Y = \sum_{p_k \in P} M(p_k) \cdot Y(p_k) \;\geq\; M(p_i) \cdot Y(p_i) + M(p_j) \cdot Y(p_j).$$

If both $M(p_1), M(p_j) \neq 0$, using the precondition of the proposition there is a contradiction. Then, either $M(p_i) = 0$ or $M(p_j) = 0$ for all the reachable markings, since there was no other restriction choosing M. $\square$

The analysis of the invariants uses markings independent of the specific firing order. Several firing orders can end with the same marking. This limitation prevents the existence of a sufficient condition for Petri Net liveness in terms of its invariants. It is possible to avoid this limitation using reachability tree analysis. As we saw previously, if the reachability tree contains any leaf node, the net is not alive. Tree search algorithms can determine the liveness property of a Net.

The next section will show an extension of the Marked Petri Nets which will allow simplifying the representation of some complex models.

4.5 Colored Petri Nets

With the objective of having simpler descriptions of complex systems, a new type of Petri Net is introduced, Colored Petri Nets [29]. They will be used as a modeling language with the capability to analyze complex systems more easily than using Marked Petri Nets. In particular, a subset of them are studied in this project, Well-formed Nets:

$$\text{Well-formed Nets} \quad \subset \quad \text{Colored Petri Nets} \quad \subset \quad \text{High Level Petri Nets}$$

Intuitively, in Colored Petri Nets, it is possible to have different mark types, each with a different color. These colors do not need to be the same in all the places. Each place can define a different color domain. If the color domain is finite, it is possible to transform the Color Petri Net into simpler Nets without colors using an unfolding procedure. If the firing of the output transitions of a place does not depend on the colors, the colors can be viewed as a way to distinguish information (i.e. each color represents the firing from a different transition) but they don't increase the expressiveness of Marked Petri Nets.

More information is required to represent different information flows. That's why the arcs between places and transitions (and only these, not the arcs between transitions and places) have annotations to restrict the marks that enable a transition. More information is required to represent different information flows. That's why the arcs between places and transitions (and only these, not from transitions to places) have annotations to restrict the marks that enable a transition.

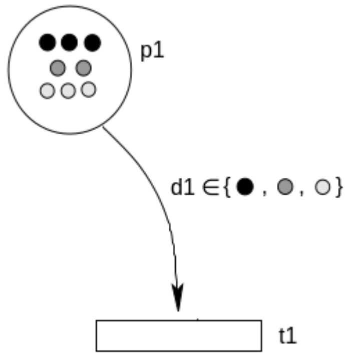


Figure 21: Colored Petri Net. Three different colors in place p_1, but only one ($d_i \in \{d_1, d_2, d_3\}$) enable transition t_1.

In Figure 21 a Colored Petri Net is shown. Marked Petri Nets can be generalized to include firing priorities, referencing different priorities drawing transitions with either boxes or lines. For simplicity, there will be no consideration for priorities. In the image, there is a label in the arc, d_i, belonging to the domain of possible colors for place p_1: $\{d_1, d_2, d_3\}$. It is clear now that the marking definition must change to represent the distribution of colors in the marks, $M = \{m_1, m_2, m_3\}$. In the image, there is a label in the arc, d_i, belonging to the domain of possible colors for place p_1: $\{d_1, d_2, d_3\}$. It is clear now that the marking definition must change to represent the distribution of colors in the marks, $M = \{m_1, m_2, m_3\}$.

Definition 29. **Colored Petri Net (CPN)** is 6-tuple $\mathscr{R} := (P,\ T,\ C^-,\ C^+,\ \mathscr{C},\ cd)$ verifying:

- $P,\ T$ are non-empty finite disjoint sets representing the places and the transitions.
- $\mathscr{C}$ is the finite set of possible color classes for the marks.
- $cd : P \vee T \to \mathscr{C}$ defines the color domain for each place or transition.
- $C^-[p,t],\ C^+[t,p] : cd(t) \to Bag(cd(p))$ are the transition matrices extending the definition of D^- and D^+. Now the color must be taken into account in the matrix definitions. The *Bag* function defines a multiset of colors, it is a set allowing several items of the same type. It allows for defining the matrices, specifying also the colors for each element.

The incidence matrix definition is extended in Colored Petri Nets:

$$C[p,t] : cd(t) \ \to \ Bag(cd(p))$$
$$C[p,t](x) := C^+[t,p](x) \ - \ C^-[p,t](x).$$

Not all the places and transitions need extra colors. The neutral color is $C_\bullet = \{\bullet\}$. If $\mathscr{C} = \{C_\bullet\}$ then the Color Petri Net can be simplified to a Marked Petri Net since

$$cd(p) = C_\bullet \quad \forall p \in P \qquad \wedge \qquad cd(t) = C_\bullet \quad \forall t \in T$$

Previously, the marking of a place was represented with a number. Now, the definition must be generalized to include the distributions of color in the marks.

Definition 30. **Marking in Colored Petri Net** is a vector $M := (M[p_1], ..., M[p_n])$ representing the distribution of colors for each place. Each element is a function

$$M[p_i] : cd(p_i) \to \mathbb{N}_0^+ \qquad p_i \in P,$$

where $M[p_i, c] := M[p_i][c]$ represents the number of marks at place p_i with color $c \in cd(p_i)$.

Definition 31. **CPN system** is a 2-tuple $< \mathscr{R},\, M_0 >$ where $\mathscr{R}$ is a colored Petri Net and M_0 is a valid initial marking.

CPN transition firings have increased granularity. A transition $t \in T$ is enabled for a color, noted as

$$< t, c > \text{ is enabled} \quad \iff \quad \forall p \in I(t) \;\; M[p,c] \geq C^-[p,t](c) \qquad c \in cd(t) \subseteq \bigcap_{p_i \in I(t)} cd(p_i).$$

The state equation is

$$M' := M + C[p,t](c) \qquad p \in P,\ t \in T,\ c \in cd(t),\ M, M' \in \mathscr{M}(R, M_0).$$

The opportunity to include any amount of colors in a place dramatically expands the possible systems to make using a small amount of places/transitions. The concept of the reachability tree is still the same, changing firing transitions with firings of the combinations of transition and colors. In practice, even for small nets, this tree could be huge because a place/transition can handle any amount of colors, and each color would create a node in the tree.

Let's analyze Figure 22. It is a simplified version of the producer-consumer problem (also shown in Figure 20). The producer would be the combination of p_1 and t_1, the buffer is p_2. It could also be considered that p_2 is the consumer, it can store any amount of marks. The transition t_2 creates two marks for each firing of t_1, one will go to p_1 and the other to p_2, the number of marks has no limit (there is no K-limitation there). In this simplified model the power of Colored Petri Nets is not shown yet. The model in Figure 22 can describe an infinite sequence of events, i.e. clicks from users without knowing the user that clicked.

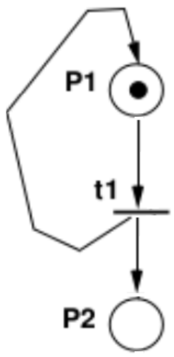


Figure 22: Modeling infinite user clicks (it is a simplified version of producer-consumer problem).

Colored Petri Nets would allow, for example, to represent a finite amount of users creating events (i.e. clicks), with the power to distinguish the origin using the color domain(s). Another possibility is to expand the complexity of user input to better represent realistic real use cases. Figure 23 represents an infinite amount of user choices (true/false) thanks to the extra expressiveness of having two colors. Now, the color domain would have two colors.

$$\mathscr{C} = \{c_1, c_2\} \qquad cd(p_1) = cd(p_2) = cd(t_1) = \{c_1, c_2\}$$

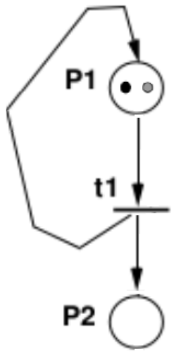


Figure 23: Modeling infinite user choices (true/false).

The generalization of the concept for storing any finite input form in Petri Nets is as follows. Imagine a field in an online form to store the first name and then register a user in a system. This kind of field usually has a maximum length (it does not make sense to allow first names with thousands of characters). Hence, there is a finite amount of characters in the form. A character encoding (i.e. UTF-8, Unicode, ...) represents each one of these characters. Therefore, a character can be encapsulated in a finite amount of bits of information. It turns out that this generalization extends to other types of data (numbers, etc.), in general, to any serializable object potentially sent in a fixed maximum amount of bits. The representation is not straightforward, data abstraction(s) and set theory should be used to simplify the definitions for complex data types.

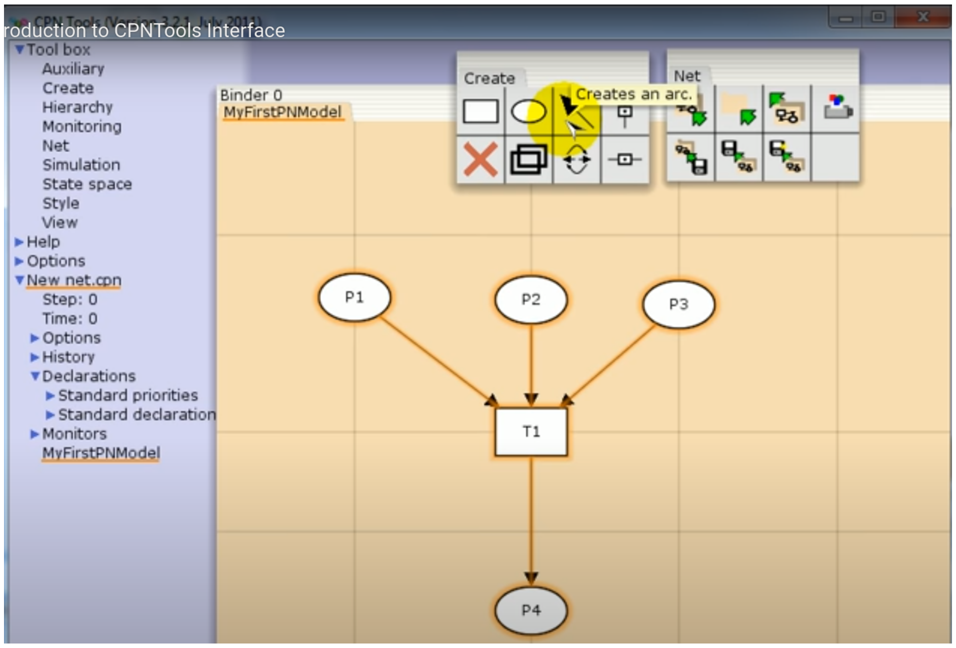


Figure 24: Simplified CPN tools interface (from [27]). Tool box for creating Colored Petri Nets.

It is not optimal nor desirable to handle all these elements by hand. To handle the increased complexity, CPN Tools are presented [1], see Figure 24. The origin of this tool is another tool (Design/CPN) from a research group from the University of Aarhus. It is a WIMP (Windows, Icons, Menus, Pointer) graphical editor interface to model Colored Petri Nets and interact with them. The first version dates from the year 2000 and requires OpenGL. [31] CPN tools interface is shown in Figure 24. Further references can be found in *https://www.cpntools.org/*.

5 Multi-tenant Web Store implementation

This general implementation of a multi-tenant web store will allow businesses to create a website with their products. The most significant parts of the system are:

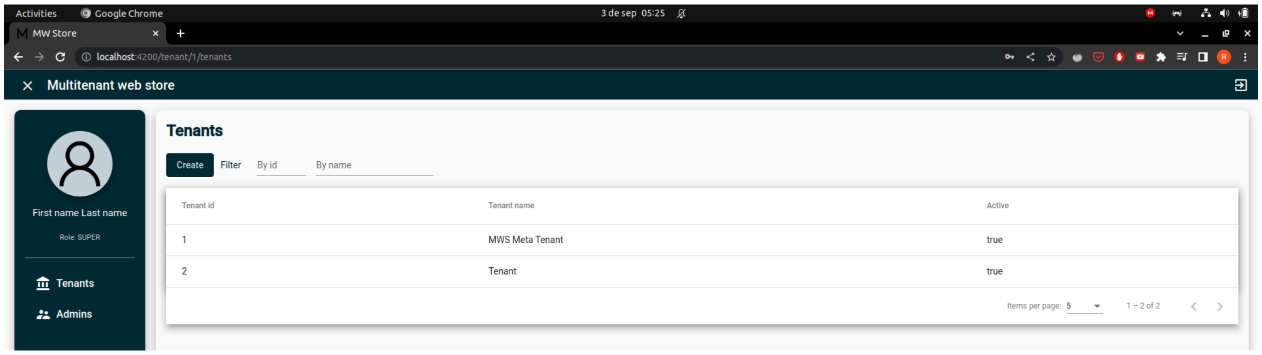
- Admin Console: here, the proprietary of the platform MW-Store handles all the tenants (in our scenario, e-commerce) and the system users. See Figures 25 and 26.
- Tenant back office: administrators manage their e-commerce configuration using this interface. It is possible to handle the products and orders here. See Figures 27 and 28.
- User front end: the client will see the available products for the e-commerce where it is registered. An end-user can purchase in different e-commerces of the MW-Store if they have distinct accounts for each e-commerce. These final users will see their order history of purchased products. See Figures 29 and 30.
- Marketing site: potential MW-Store clients (businesses) will see an overview of a generated e-commerce with commercial purposes.

In MW-Store, all these components share an interface. Each component is differentiated using automatic web URL redirection. Data isolation between tenants is achieved using the DAO pattern with default tenant filtering. The server sends a temporal encrypted tenant identifier from the back-end to the front-end stored in a JWT token generated after a valid login. An HTTP interceptor in the front end sends the token in all the requests for proper tenant identification. Sending another tenant in the request is equivalent to breaking the token encryption, fake user identifiers, and bypassing all the internal data consistency checks in the back-end.

The application verifies multi-tenancy properties. The tenants, in this case, e-commerces, share the application context and the database, having a tenant identifier for each table. The initial resource sharing approach is Shared Database, Shared Schema. In the future, if the application grows enough and the load on one database is a bottleneck, a migration to a different approach would be considered. No tenant has access to the data of other tenants by design, the front-end does not decide the tenant to query. Different tenants have different redirections to differentiate one e-commerce from the other ones. If a tenant, in some way, tries to use its credentials to access data from other tenants it will receive an error from the back-end. The nature of this application is inherently concurrent and several properties will be formalized.

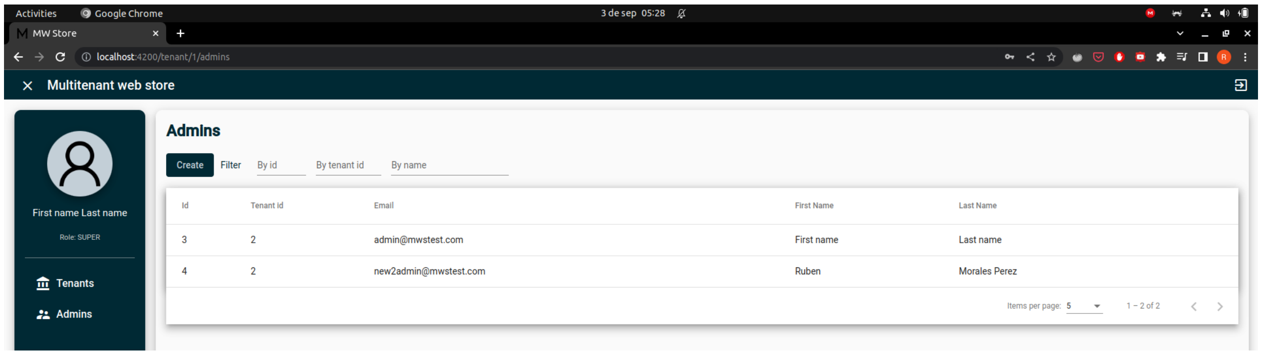
5.1 Functional requirements

First, let's discuss the structure of the system. The root of the system is the super user. This user has access privileges to the MW-Store admin console, where all tenants (Figure 25) and their respective admins (Figure 26) are managed.



Tenant id	Tenant name	Active
1	MWS Meta Tenant	true
2	Tenant	true

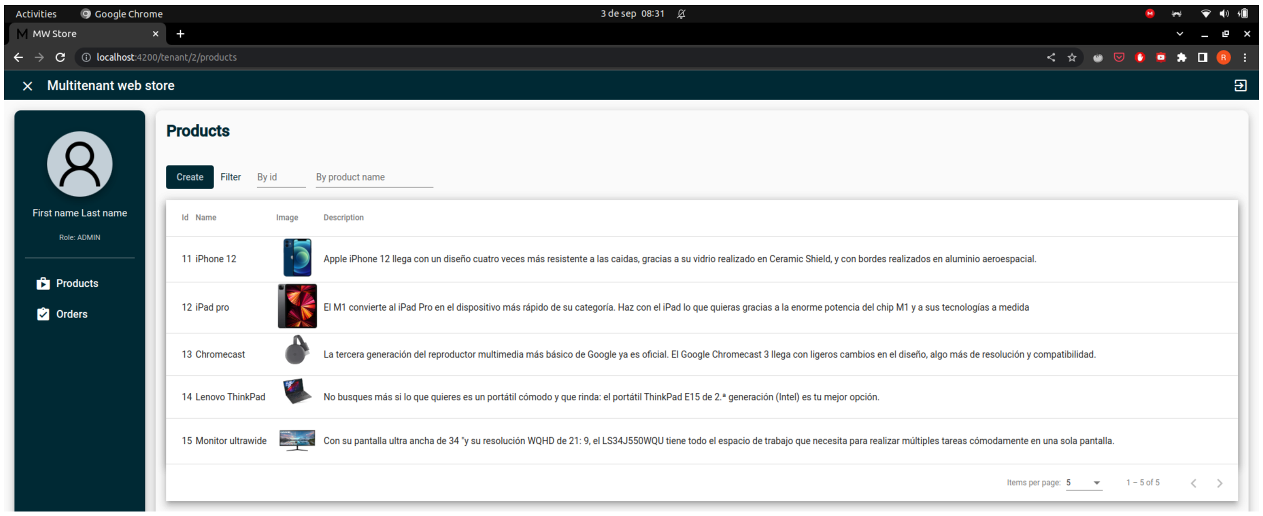
Figure 25: MW-Store admin console to manage tenants.



Id	Tenant id	Email	First Name	Last Name
3	2	admin@mwstest.com	First name	Last name
4	2	new2admin@mwstest.com	Ruben	Morales Perez

Figure 26: MW-Store admin console to manage site admins.

After the tenant and, at least, one admin for the tenant, products should be created. The new admin has access to a back office with product management (Figure 27) and access to all the orders of the tenant (Figure 28).


Activities Google Chrome 3 de sep 08:31

MW Store

localhost:4200/tenant/2/products

Multitenant web store

First name Last name

Role: ADMIN

Products

Orders

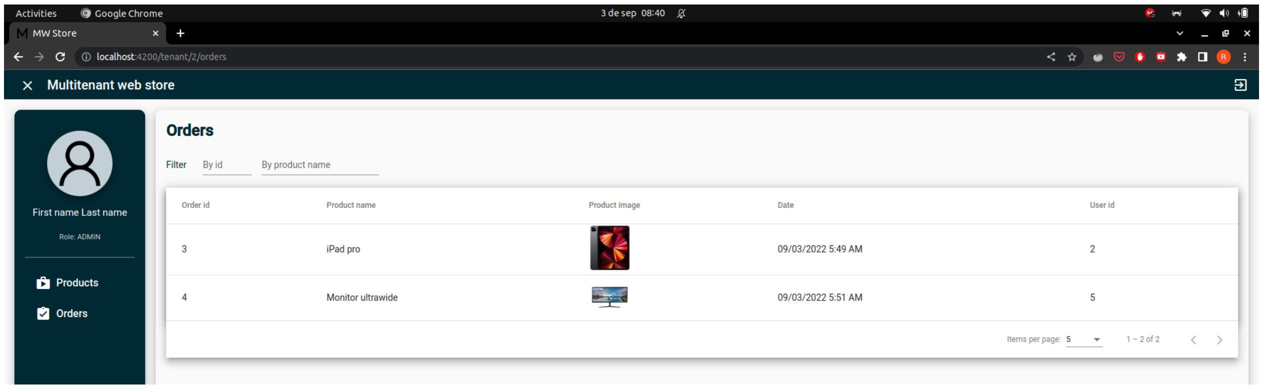
Products

Create Filter By id By product name

Id	Name	Image	Description
11	iPhone 12		Apple iPhone 12 llega con un diseño cuatro veces más resistente a las caidas, gracias a su vidrio realizado en Ceramic Shield, y con bordes realizados en aluminio aeroespacial.
12	iPad pro		El M1 convierte al iPad Pro en el dispositivo más rápido de su categoría. Haz con el iPad lo que quieras gracias a la enorme potencia del chip M1 y a sus tecnologías a medida
13	Chromecast		La tercera generación del reproductor multimedia más básico de Google ya es oficial. El Google Chromecast 3 llega con ligeros cambios en el diseño, algo más de resolución y compatibilidad.
14	Lenovo ThinkPad		No busques más si lo que quieres es un portátil cómodo y que rinda: el portátil ThinkPad E15 de 2.ª generación (Intel) es tu mejor opción.
15	Monitor ultrawide		Con su pantalla ultra ancha de 34 "y su resolución WQHD de 21: 9, el LS34J550WQU tiene todo el espacio de trabajo que necesita para realizar múltiples tareas cómodamente en una sola pantalla.

Items per page: 5 1 – 5 of 5

Figure 27: Tenant admin back office. Products management interface.


Activities Google Chrome 3 de sep 08:40

MW Store

localhost:4200/tenant/2/orders

Multitenant web store

First name Last name

Role: ADMIN

Products

Orders

Orders

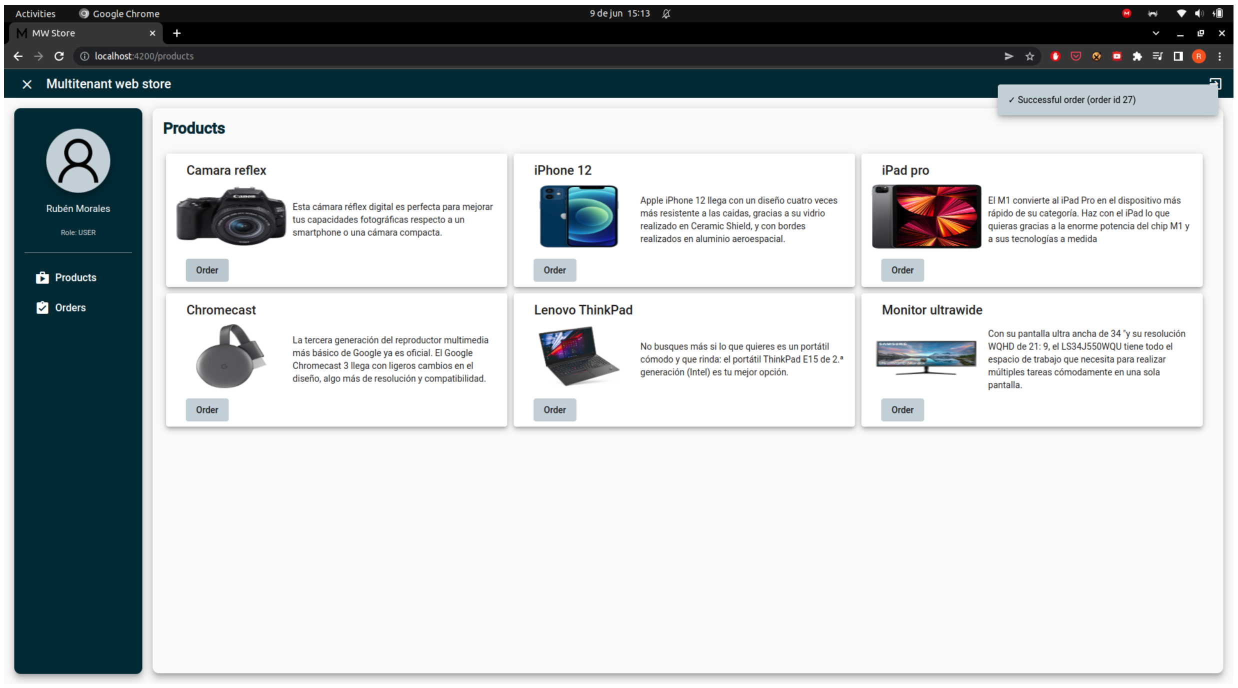
Filter By id By product name

Order id	Product name	Product image	Date	User id
3	iPad pro		09/03/2022 5:49 AM	2
4	Monitor ultrawide		09/03/2022 5:51 AM	5

Items per page: 5 1 – 2 of 2

Figure 28: Tenant admin back office. Order management interface.

The designed system will store products, each one associated to a specific tenant. Different tenants are allowed to create different products (Figure 29) that will be bought by end users (Figure 30). Each tenant is an organization and will handle their product(s) and keep track of the orders.



Activities   Google Chrome   9 de jun 15:13

MW Store

localhost:4200/products

Multitenant web store

✓ Successful order (order id 27)

Rubén Morales

Role: USER

Products

Orders

Products

Camara reflex

Esta cámara réflex digital es perfecta para mejorar tus capacidades fotográficas respecto a un smartphone o una cámara compacta.

Order

iPhone 12

Apple iPhone 12 llega con un diseño cuatro veces más resistente a las caidas, gracias a su vidrio realizado en Ceramic Shield, y con bordes realizados en aluminio aeroespacial.

Order

iPad pro

El M1 convierte al iPad Pro en el dispositivo más rápido de su categoría. Haz con el iPad lo que quieras gracias a la enorme potencia del chip M1 y a sus tecnologías a medida

Order

Chromecast

La tercera generación del reproductor multimedia más básico de Google ya es oficial. El Google Chromecast 3 llega con ligeros cambios en el diseño, algo más de resolución y compatibilidad.

Order

Lenovo ThinkPad

No busques más si lo que quieres es un portátil cómodo y que rinda: el portátil ThinkPad E15 de 2.ª generación (Intel) es tu mejor opción.

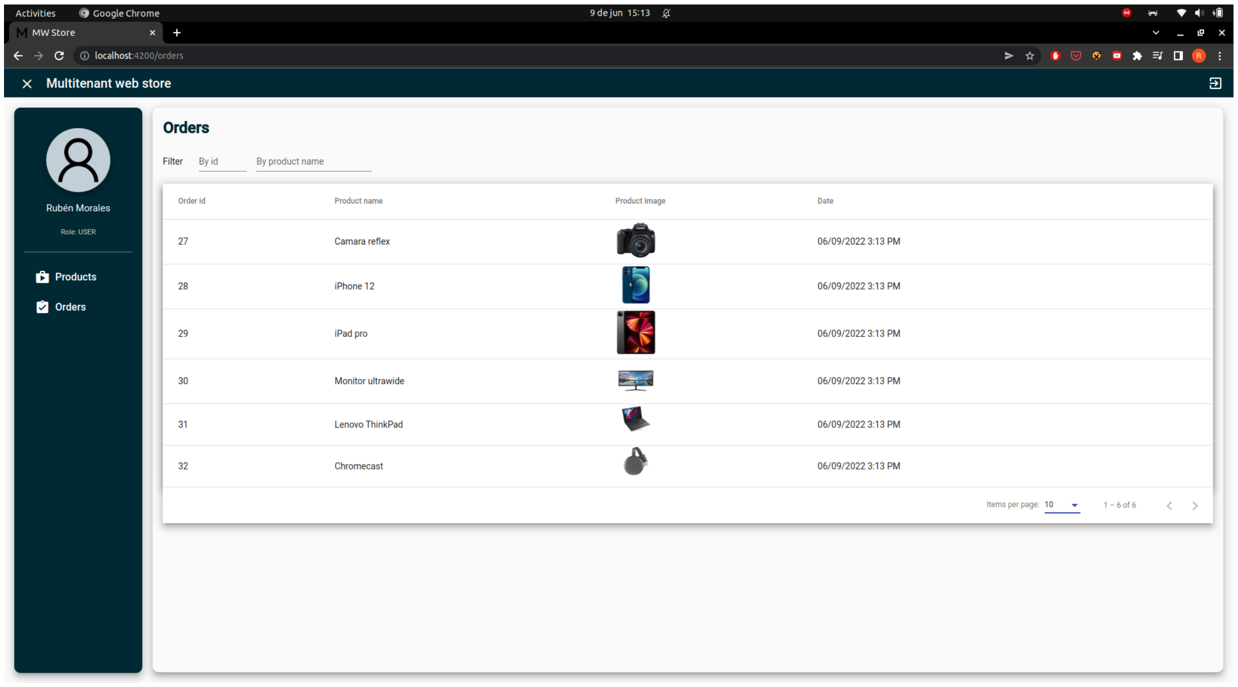
Order

Monitor ultrawide

Con su pantalla ultra ancha de 34 "y su resolución WQHD de 21: 9, el LS34J550WQU tiene todo el espacio de trabajo que necesita para realizar múltiples tareas cómodamente en una sola pantalla.

Order

Figure 29: Visualization of the products available for one tenant.



Activities   Google Chrome   9 de jun 15:13

MW Store

localhost:4200/orders

Multitenant web store

Rubén Morales

Role: USER

Products

Orders

Orders

Filter   By id   By product name

Order Id	Product name	Product Image	Date
27	Camara reflex		06/09/2022 3:13 PM
28	iPhone 12		06/09/2022 3:13 PM
29	iPad pro		06/09/2022 3:13 PM
30	Monitor ultrawide		06/09/2022 3:13 PM
31	Lenovo ThinkPad		06/09/2022 3:13 PM
32	Chromecast		06/09/2022 3:13 PM

Items per page: 10   1 – 6 of 6

Figure 30: Visualization of the orders available for one tenant.

The technological stack of the system is:

- Front end: Angular 13.2.7 (testing with Karma + Jasmine)
- Back end: Java 17 (testing with JUnit).
- Database: agnostic database calls using Hibernate as ORM. (several databases were used as example).

The minimal database structure to handle the e-commerce is shown in Figure 31.

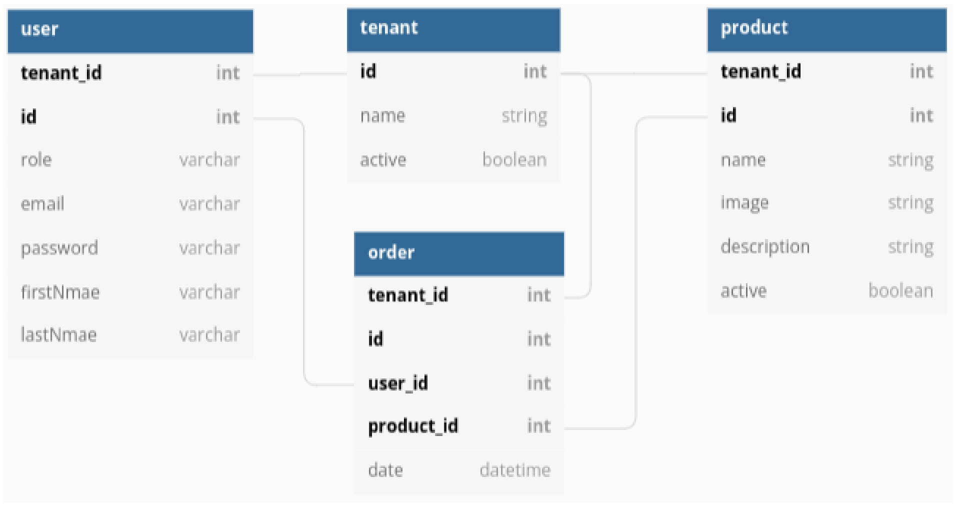


Figure 31: Multi-tenant e-commerce minimal database structure.

5.2 Colored Petri Net model

It turns out that the model will be a special kind of producer-consumer (like Figure 23, modeling an infinite amount of user choices). The model heavily depends on the specifications of the system. Functional specifications determine the properties of the final CPN. The simplest general multi-tenant e-commerce architecture is used for a better understanding of its properties. Even with a minimalist model, the resulting Colored Petri Net is not straightforward. There are several parts of this multi-tenancy e-commerce:

- Creation of tenants (businesses).
- Creation of final users (buying the products).
- Creation of products (by the tenants).
- Creation of orders (by the users, with only one product).

There are other important properties to model the system:

- A tenant must exist to be able to create a user (linked to that tenant).
- Only a tenant can create a product.

- A product without any associated order and a user must exist to create an order.

Let's start with the modeling of tenant creation.

5.2.1 Tenants sub-net

First, let's create a producer-consumer subnet to represent the tenant creation. One possible problem of some producer-consumer models with Colored Petri Net is producing an infinite amount of marks. The possibility of having infinite marks in a place will prevent K-limitation, and it is not a desirable property. A property to allow the creation of infinite marks is allowing the production of the same mark an indefinite number of times. The proposed model to solve this is to split the creation of marks between two places: one place with the finite possible values of not created entities and another with the created ones. These two places are linked using a transition.

There are unique constraints when creating a tenant in the database. The name of the tenant is a possible unique identifier when it is not possible to change it. To address this issue, the subnet of tenant creation will have three parts (see Figure 32):

- Place P_NT: the set of all possible tenants not yet created. Each color represents one tenant.
- Transition T_T: represents the creation of a new tenant. The transition color domain derives from its input transition.
- A place, P_T, containing the created tenants.
- An arc from P_NT to T_T, with the set of tenants as input.
- An arc from T_T to P_T with the set of tenants as output.

Another possible way to model the system is using P_NT as all the possible input values and the transition subset as the valid ones (i.e., tenant names with a maximum length, etc.). In this model, only one set is used, just for simplicity in the mathematical model.

The Figure 32 contains a simplified version of the model for tenant creation. There is only one initial mark in P_NT, representing a possible name for tenant creation using T_T. The reason to represent the set $TenantSet$ as $\{'e-commerce'\}$ is just simplicity in the view. Indeed, there would be as many marks as tenant names with less or equal than ten characters. The tenant sub-net is colored in red to simplify the global picture.

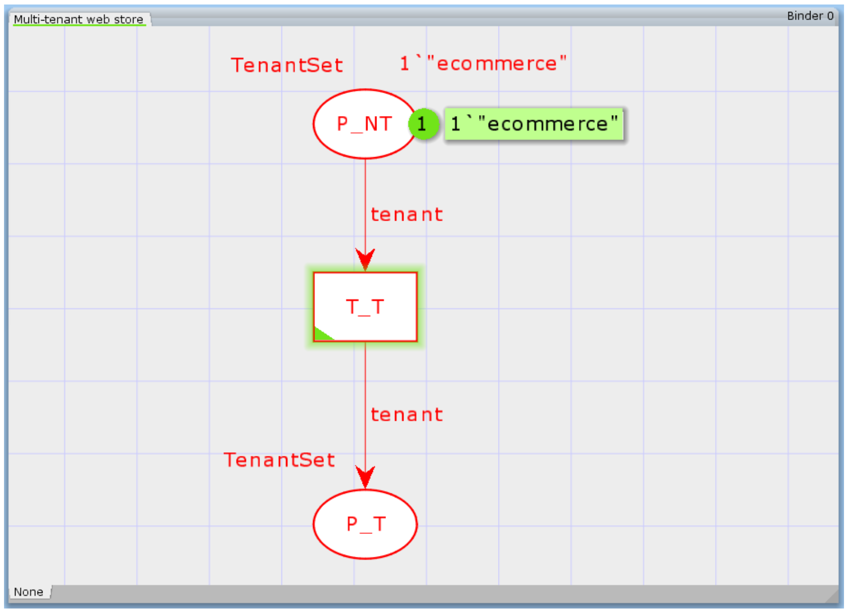


Figure 32: CPN for tenant creation. Initial non created tenant 'e-commerce'.

Formally, the sub-net is defined as $TENANT\_SUBNET := (P, T, C^-, C^+, \mathscr{C}, cd)$ where

- $P = \{P\_NT,\ P\_T\}$
- $T = \{T\_T\}$
- $(C) = \{TenantSet\}$ where $TenantSet := \{a|...|z\}^{\{1|...|10\}}$
- $cd : P \vee T \rightarrow \mathscr{C}$ where $cd(P\_NT) = cd(P\_T) = cd(T\_T) = TenantSet$
- $C^-[p,t] = cd(t) \rightarrow Bag(cd(p))$

$$C^-[p,t] = \begin{cases} \{TenantSet\} & if\ (p = P\_NT\ \wedge\ t = T\_T) \\ \{\} & otherwise \end{cases}$$

- $C^+[t,p] = cd(t) \rightarrow Bag(cd(p))$

$$C^+[t,p] = \begin{cases} \{TenantSet\} & if\ (t = T\_T\ \wedge\ p = P\_T) \\ \{\} & otherwise \end{cases}$$

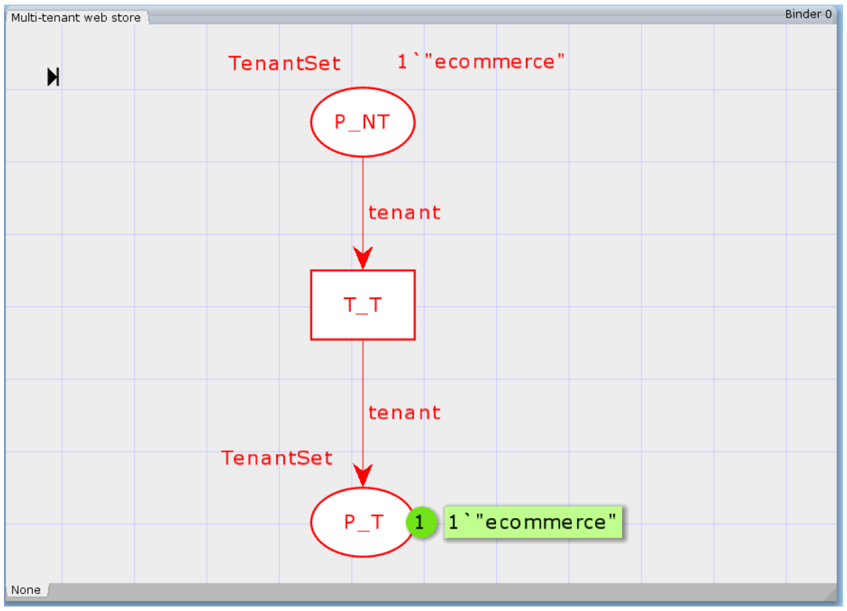


Figure 33: CPN after tenant creation. Now, there is a tenant 'e-commerce' as a mark in P_T

The current firing sequence is $\{T\_T\}$. The next step is to expand this system to allow user creation.

5.2.2 Users sub-net

It turns out that a similar (slightly modified) structure should be valid for user creation. In multi-tenancy, an end user is created just for a specific tenant. In this scenario, an end-user depends on the desired web store business. The same person could have several end-users in the system, one (or more) for each web store. An identical set to $TenantSet$, $UserSet$ stores the possible unique names of the users.

In Figure 34, using the same set for P_NU and P_U prevents us from having users with the same name in different tenants. This limitation is artificial and used for simplicity in the model. Indeed, the correct output set from T_U would be $TenantSet \times UserSet$ because the tenant information is essential and must be stored. Due to the limitations of CPNTools, it is impossible to define the Cartesian product of two large sets as an output set: "If a free variable from a large color set is bound to an output arc, an error message will indicate the problem." [28].

It is possible to skip this limitation by defining an artificial Cartesian product. The length limit for tenant names was ten, idem for users. If $UserSet$ length is extended then we can represent a mark as "user@tenant". If the arc to create users limits properly the length (allowing only a subset of its input set) the result is equivalent to the Cartesian product. Anyway, this trick is just useful for educational purposes, showing the user associated with the tenant in the mark.

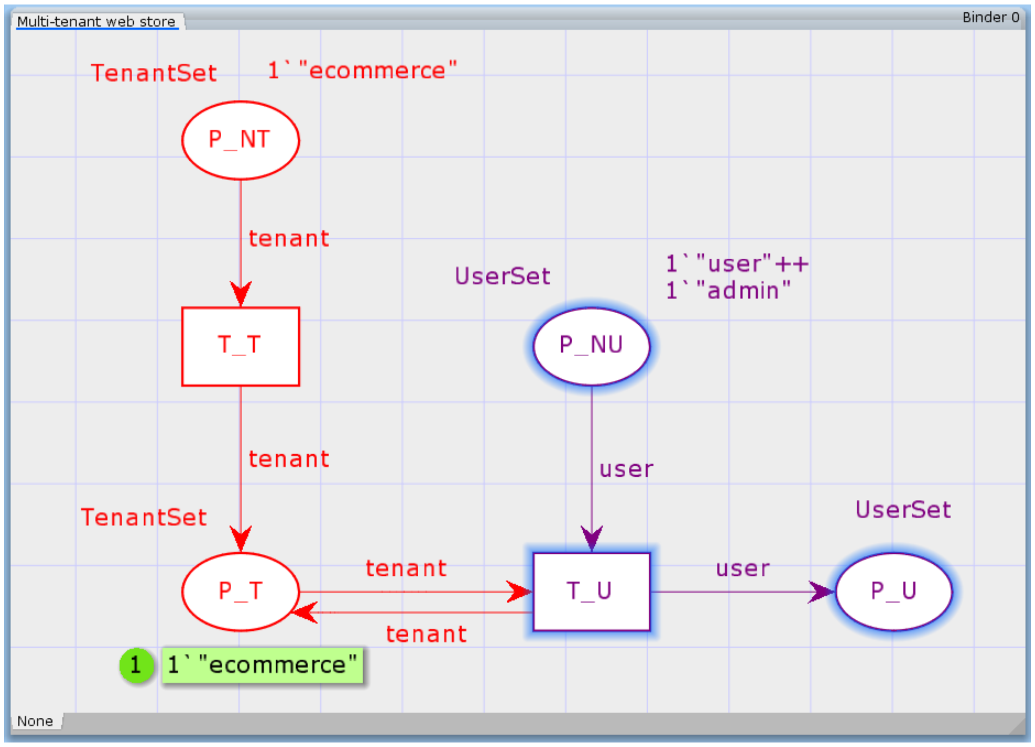


Figure 34: CPN able to create end users.

It is remarkable to notice the two-arc between the *TenantSet* and the transitions to create users (Figure 34). Losing the tenant after the user creation is not a good idea. The model must represent a system with the possibility of creating several (but a finite number of) users!

This approach has a problem, though. It is incomplete because there are three kinds of users:

- Super: the owner(s) of Multi-tenant Web Store. They have permission to control the tenants. The super user is linked to a special kind of tenant which controls other tenants' metadata. There could be several super users, but only a super user can create other super users.
- Admin: administrator(s) of each business. They have a custom control panel where products are created and orders monitored. Only the super user can create new admins.
- User: end-user buying the products of one specific web store. There is no restriction on the creation of end users rather than unique name constraints. Anyone can be a potential user of each e-commerce. For that reason, in this model, anyone can create an end user for any business.

Three more transitions are needed to differentiate the control flow, one per user type. The user management subnet is purple. Let's see how to create the different users, considering the restrictions specified above.

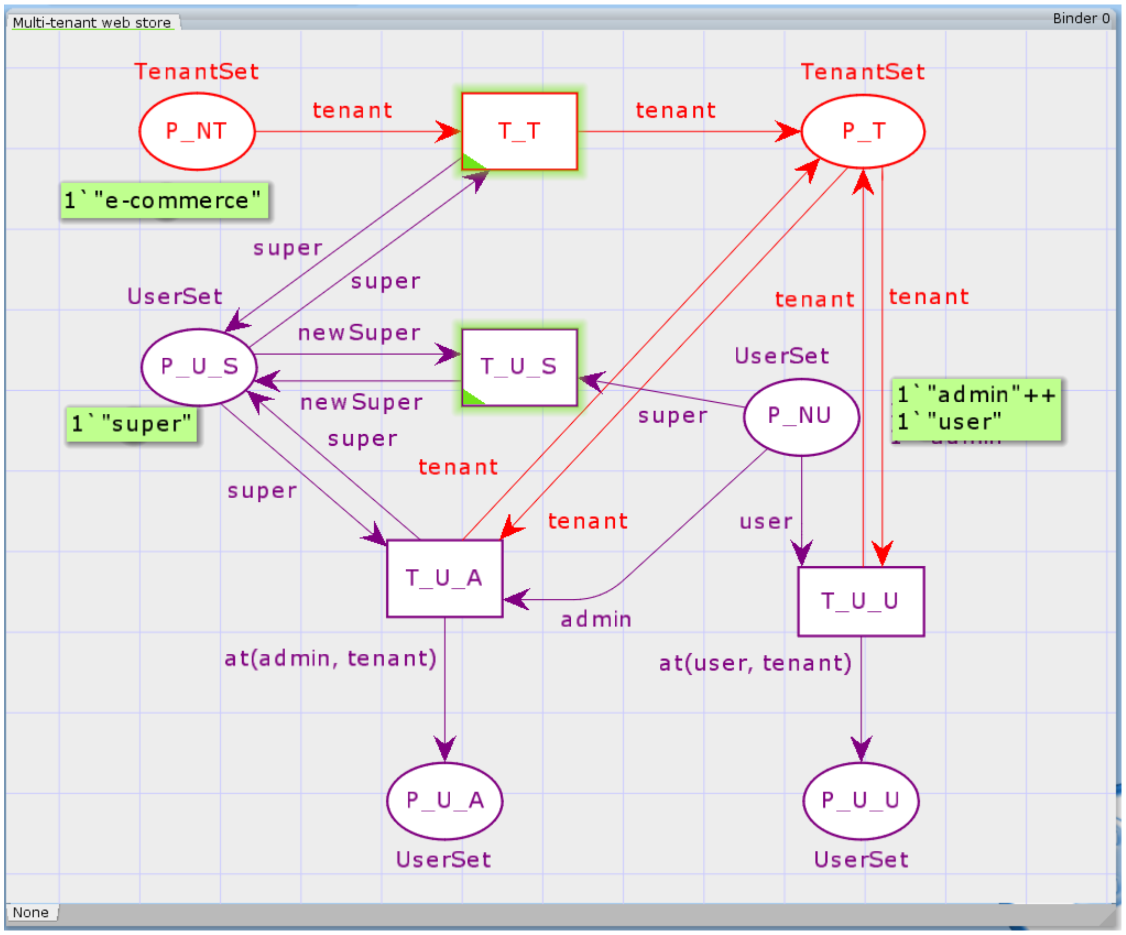


Figure 35: CPN with all the user's creation logic at the initial marking

An important property to consider now is that the system's initial data changed. It is not the tenant anymore but the initial super user (see Figure 35). The super user is the only initial data in the data store at the beginning. This super user has the power to create tenants on demand. There is no magic entity creating tenants but an actual operator (super user) from MWS. It is debatable if the tenant should also be in the initial system's data. It is a good practice to minimize the direct manipulation of the database. Having only the super user as the initial data allow tenant creation with the possible security constraints checked at the application level. The super user as the only initial data seems like a reasonable idea. The only user allowed to create super users and tenants is the super user.

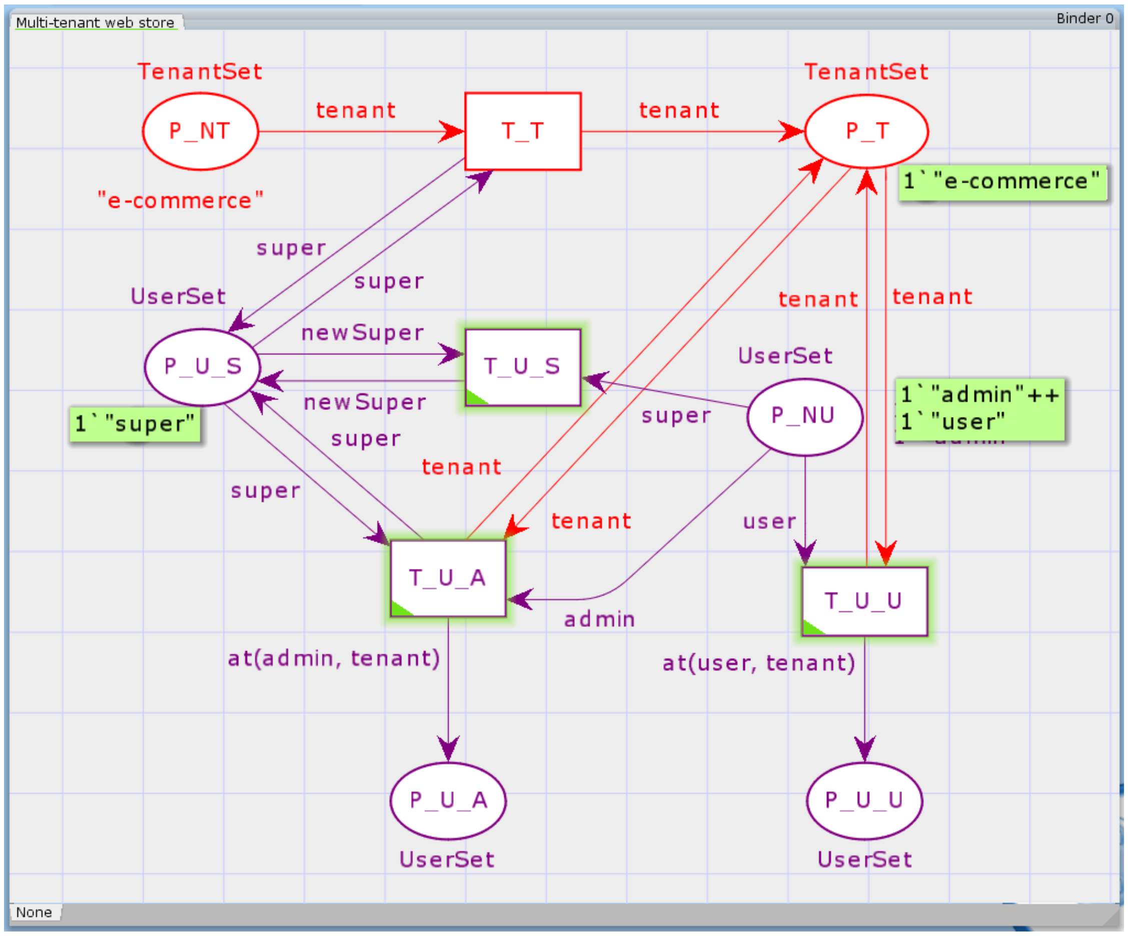


Figure 36: CPN with a super user and a tenant created

Now, the way to create tenants has changed to integrate its logic with the user management (see Figure 36). The super user already created a new e-commerce tenant. After that, the super user can still create more super users because there are no other restrictions for that. The difference with the previous step is the possibility to create admins. The admins depend on a tenant to make sense.

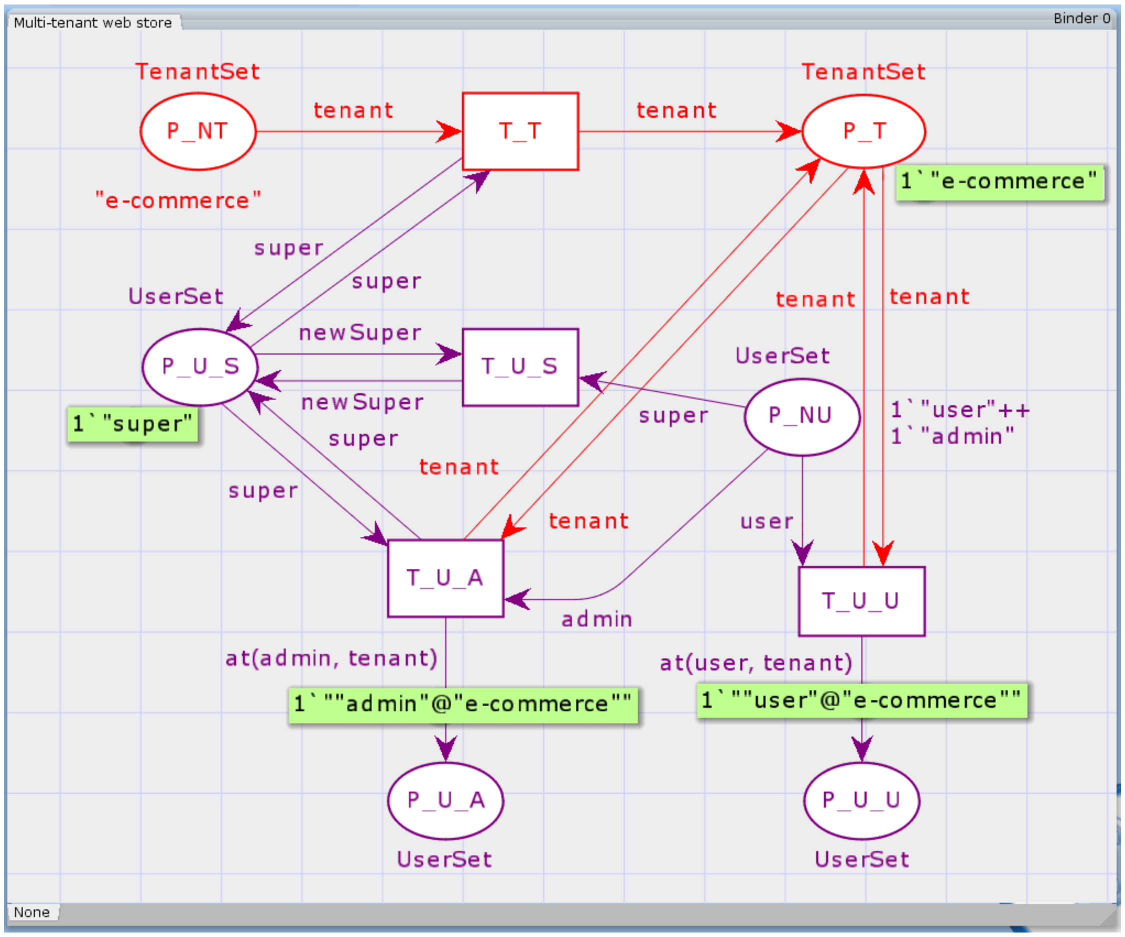


Figure 37: CPN after all users created

In Figure 37 the admin and one user of the e-commerce were created. There is no enabled transition now because there are no more marks in *P_NU* to create super, admins, or end users. The model will become more complex, so, from now on, we will consider only one tenant, one super user, one admin, and one end user.

The previous model(s) changed. Now, the definition of the net in Figure 37 is:
$USERS\_SUBNET := (P, T, C^-, C^+, \mathscr{C}, cd)$,

- $P = \{P\_NT,\ P\_T\ \ P\_U\_S,\ P\_U\_A,\ P\_U\_U\}$
- $T = \{T\_T,\ T\_U\_S,\ T\_U\_A,\ T\_U\_U\}$
- $(C) = \{TenantSet, UserSet\}$ where $TenantSet = UserSet = ASCII^{\{1|...|10\}}$ where $ASCII$ is the set of possible characters in the ASCII table. The sets $TenantSet$ and $UserSet$, even though they have the same elements, are treated differently to understand the model relationships.

- $cd : P \vee T \to \mathscr{C}$ where
  $cd(P\_NT) = cd(P\_T) = cd(T\_T) = TenantSet$,
  $cd(P\_U\_S) = cd(P\_U\_A) = cd(P\_U\_U) = cd(P\_NU)$
  $= cd(T\_U\_S) = cd(T\_U\_A) = cd(T\_U\_U) = UserSet$.

- $C^-[p,t] = cd(t) \to Bag(cd(p))$
  $$C^-[p,t] = \begin{cases} \{TenantSet\} & if\ (p = P\_NT\ \wedge\ t = T\_T) \\ \{UserSet\} & if\ (t \in \{T\_U\_S,\ T\_U\_A,\ T\_U\_U\}) \\ \{\} & otherwise \end{cases}$$

- $C^+[t,p] = cd(t) \to Bag(cd(p))$
  $$C^+[t,p] = \begin{cases} \{TenantSet\} & if\ (t = T\_T\ \wedge\ p = P\_T) \\ \{UserSet\} & if \begin{cases} t \in \{T\_U\_S,\ T\_U\_A,\ T\_U\_U\} \\ \wedge \\ p \in \{P\_T,\ P\_U\_S,\ P\_U\_A,\ P\_U\_U\} \end{cases} \\ \{\} & otherwise \end{cases}$$

The current firing sequence is

- T_T: Creation of a tenant.
- T_U_A: Creation of an admin for the new tenant.
- T_U_U: Creation of the first final user of the system, a client of the created tenant.

The next step is to integrate the product creation.

5.2.3 Products sub-net

The products sub-net follows the same principles of the previous models for producer-consumer. The proposed subnet (in brown) for product creation is shown in Figure 38.

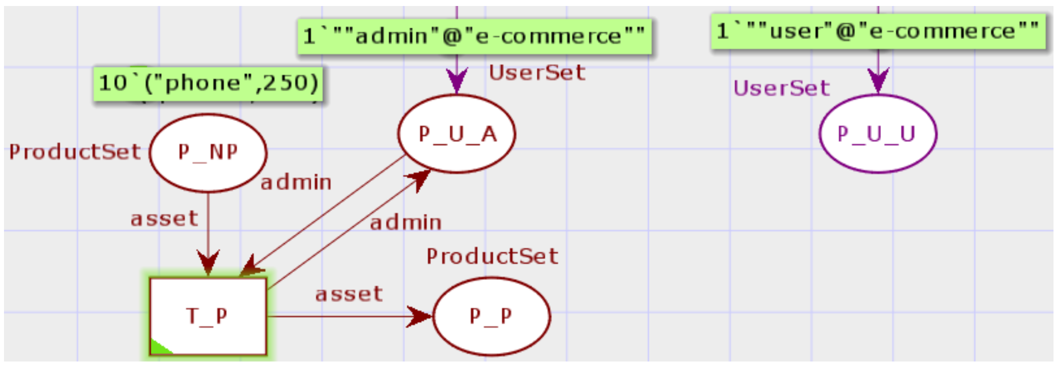


Figure 38: CPN for product creation.

The model from Figure 38 has a relevant limitation. Even the most basic web store needs the product price and the ability to change it. The modified net for product management is in Figure 39.

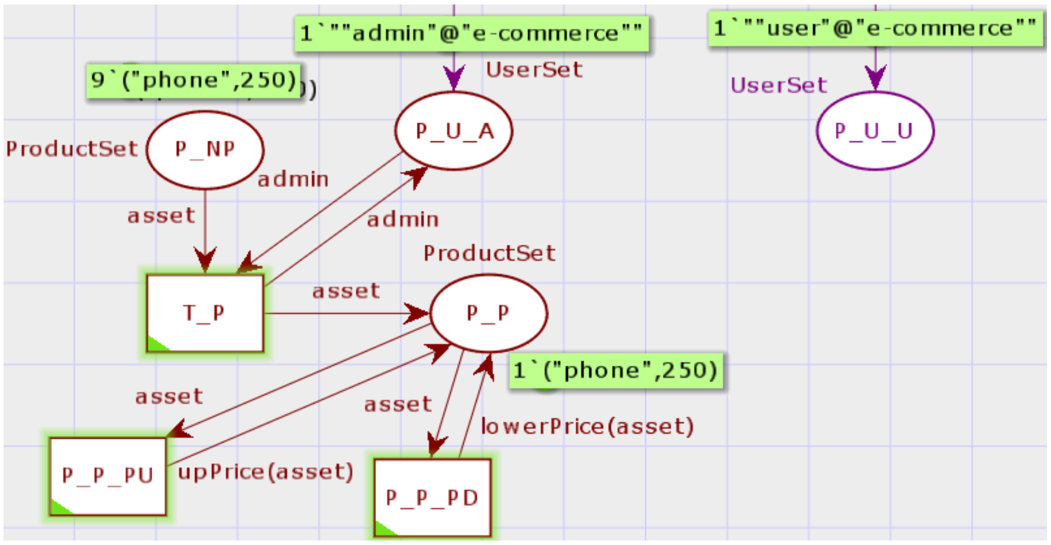


Figure 39: CPN with product prices creation and update.

In Figure 39 there are some things that can be improved. First, the limitation of incrementing or decrementing the product price one by one does not make sense. A delta value simplifies the logic, not requiring several transition firings to change the price. The possible delta values set must be a finite subset of $\mathbb{R}$. Otherwise, it would not meet the definitions.

A user might be involved in a product's price update. Final users must not change the price. For security reasons, just the tenant admin of a tenant will be able to manage the product prices (not other tenant products). This restriction will be verified using an intermediate transition between the place of tenant admins and the place of products. It is possible (not implemented for simplicity) to make granular permissions to restrict product updates just for a subset of admins.

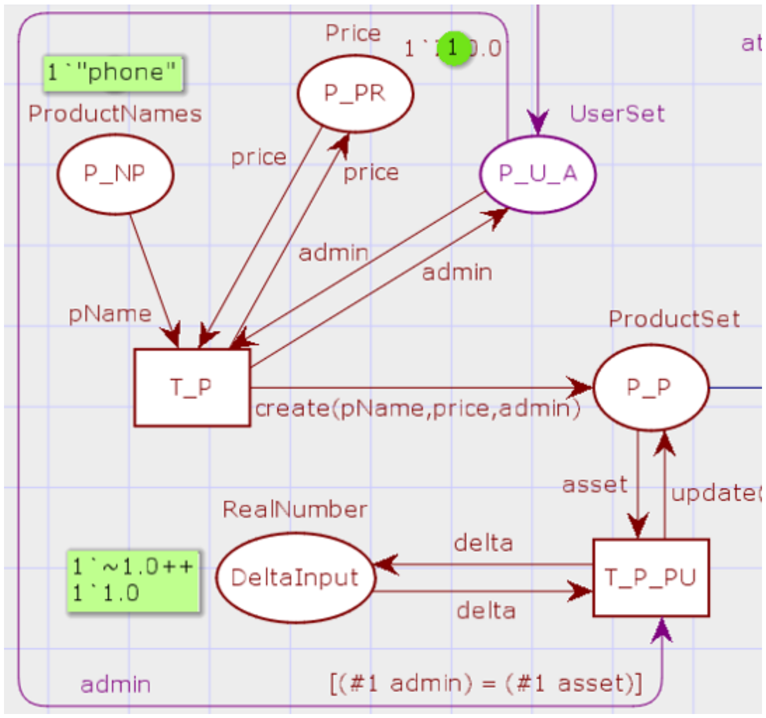


Figure 40: CPN with improved product management

The model will use the improved product management model from Figure 40. The product price is updated using a defined method $update(asset, delta)$ that computes the sum of the values. The idea is to replace a product with its new version in one atomic transaction using transition T_P_PU. It is the mission of the last subnet, the modeling of product orders.

5.2.4 Orders sub-net

An order in this system is a confirmation of a user action requesting a product. The assumptions are:

- Any available product in their tenant's store can be bought by any user, regardless of its ongoing price.
- An order is limited to one product.
- There is no limitation on the number of products to order per user.
- After order creation, the product will not be available to another end user.
- A user can buy any amount of orders.

The stock of a product can be simulated shifting the complexity to the data model in P_NP. There will be no lock to buy a product. In real life, it is common to prevent product sales to external users when the product is already in other user cart.

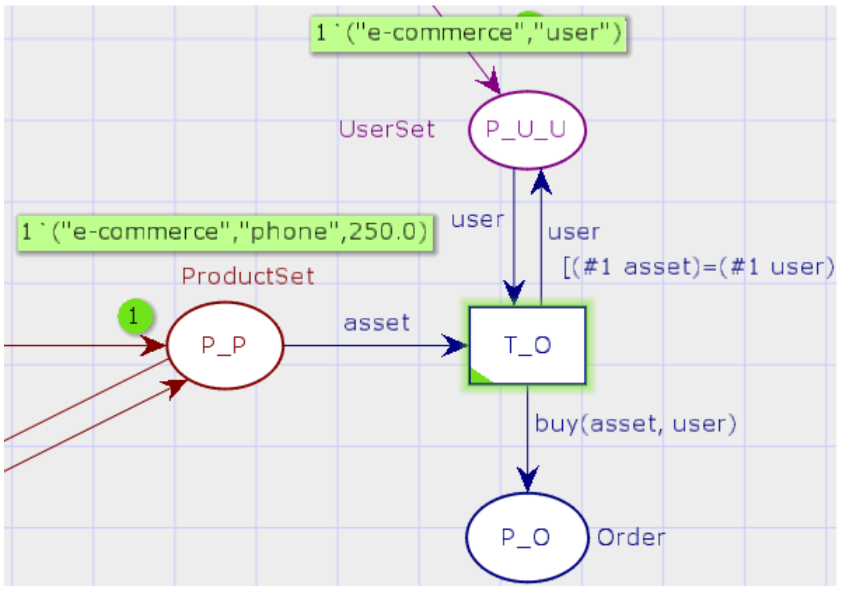


Figure 41: CPN to create orders for products.

Figure 41 show the order management system. If a user (mark in *P_U_U*) and a product (mark at *P_P*) coexist, then the transition to order products, *P_O*, is enabled. Transition *T_O* is the last element of the net. It serves as the order history of all users.

5.3 MWS Colored Petri Net Analysis

Finally, Figure 42 shows the final Colored Petri Net. It models the Multi-tenant Web Store with an order finished and kept in the order history (place *T_O*).

In summary, now that the whole CPN is defined, here are the minimal transition firings to end ordering a product in MWS.

- The initial super user exists, firing *T_T* will create a tenant ('e-commerce') indefinitely at *P_T*.
- The *T_U_A* transition depends on the super user and the tenant. Its firing will create an admin for the tenant at *P_U_A*.
- The independent firing of *T_U_U* creates a user for the tenant at *P_U_U*.
- The admin in *P_U_A* verifies its existence firing the transition *exists*.
- After that, the admin creates a product by firing *T_P*, storing the new product in *P_P*.

- The product at P_P and the end user at P_U_U enable the transition P_O. P_O represents the function to buy a product. Its firing will create a purchase record at T_0.

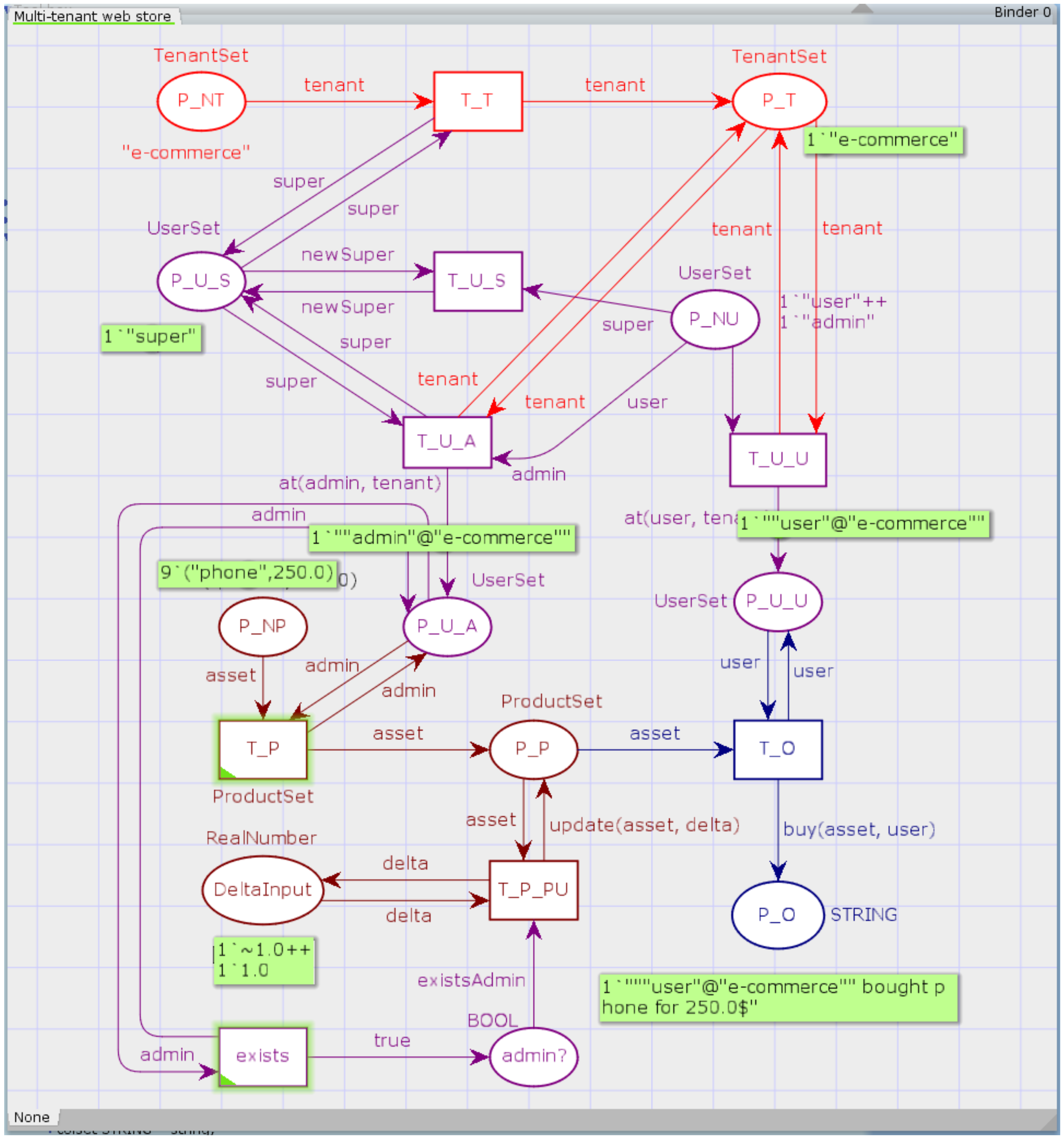


Figure 42: Multi-tenant Web Store model with a Colored Petri Net.

Formally, the definition of the sub-net in Figure 42 is:

$MWS\_CPN := (P, T, C^-, C^+, \mathscr{C}, cd)$,

- $P := \{P\_NT,\ P\_T\ P\_NU,\ P\_U\_S,\ P\_U\_A,\ P\_U\_U,\ admin?,\ P\_NP,\ DeltaInput,\ P\_P,\ P\_O\}$
- $T := \{T\_T,\ T\_U\_S,\ T\_U\_A,\ T\_U\_U,\ exists,\ T\_P,\ T\_P\_PU,\ T\_O\}$
- $\mathscr{C} := \{TenantSet,\ UserSet,\ ProductSet,\ BOOL,\ STRING,\ RealNumber,\ UserSet \times ProductSet,\ STRING \times RealNumber, RealNumber \times ProductSet \times BOOL\}$ where
  - $BOOL := \{true, false\}$
  - $TenantSet = UserSet = ProductSet = STRING = ASCII^{\{1|\dots|100\}}$ where $ASCII$ is the set of possible characters in the ASCII table. These sets, even though they have the same elements, are treated differently to understand the model relationships.
  - $RealNumber$ is $\mathbb{R} \cap [-100, 100]$ with a maximum number of ten decimals. The purpose of this limitation into a finite set is to have the possibility to unfold the net into a simpler General Petri Net. It does not include a restriction on the possible uses of updating the price of a product. There are no prices with more than ten decimals, and it is always possible to fire the same transition several times if the upper/lower bounds are insufficient.
- $cd : P \vee T \to \mathscr{C}$ where
  - $cd(P\_NT) = cd(P\_T) = cd(T\_T) = TenantSet$
  - $cd(P\_U\_S) = cd(P\_U\_A) = cd(P\_U\_U) = cd(P\_NU) = cd(T\_U\_S) = cd(T\_U\_A) = cd(T\_U\_U) = c(exists) = UserSet$
  - $cd(P\_NP) = cd(T\_P) = cd(P\_P) = ProductSet$
  - $cd(admin?) = BOOL$
  - $cd(T\_O) = STRING$
  - $cd(DeltaInput) = RealNumber$
  - $cd(P\_O) = UserSet \times ProductSet$
  - $cd(P\_NP) = STRING \times RealNumber$
  - $P\_P\_PU = RealNumber \times ProductSet \times BOOL$
- $C^- : P \times T \to Bag(cd(P))$ is a matrix of functions:

$$\begin{array}{cccccccccc}
\text{T\_T} & \text{T\_U\_S} & \text{T\_U\_A} & \text{T\_U\_U} & \text{exists} & \text{T\_P} & \text{T\_P\_PU} & \text{T\_O} & \\
\left(\begin{array}{cccccccc}
IDTS & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & IDTS & IDTS & 0 & 0 & 0 & 0 \\
0 & IDUS & IDUS & IDUS & 0 & 0 & 0 & 0 \\
IDUS & IDUS & IDUS & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & IDUS & IDUS & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & IDUS \\
0 & 0 & 0 & 0 & 0 & 0 & IDB & 0 \\
0 & 0 & 0 & 0 & 0 & IDSR & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & IDR & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & IDPS & IDPS \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right) & & & & & & & & \begin{array}{c}\text{P\_NT}\\ \text{P\_T}\\ \text{P\_NU}\\ \text{P\_U\_S}\\ \text{P\_U\_A}\\ \text{P\_U\_U}\\ \text{admin?}\\ \text{P\_NP}\\ \text{DeltaInput}\\ \text{P\_P}\\ \text{P\_O}\end{array}
\end{array}$$

- $C^+ : T \times P \to Bag(cd(P))$:

$$\begin{array}{cccccccccc}
\text{T\_T} & \text{T\_U\_S} & \text{T\_U\_A} & \text{T\_U\_U} & \text{exists} & \text{T\_P} & \text{T\_P\_PU} & \text{T\_O} & \\
\left(\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & IDTS & IDTS & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
IDUS & IDUS & IDUS & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & ATAT & 0 & 0 & IDUS & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & IDUS \\
0 & 0 & 0 & 0 & exists & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & IDR & 0 \\
0 & 0 & 0 & 0 & 0 & IDPS & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & buy
\end{array}\right) & & & & & & & & \begin{array}{c}\text{P\_NT}\\ \text{P\_T}\\ \text{P\_NU}\\ \text{P\_U\_S}\\ \text{P\_U\_A}\\ \text{P\_U\_U}\\ \text{admin?}\\ \text{P\_NP}\\ \text{DeltaInput}\\ \text{P\_P}\\ \text{P\_O}\end{array}
\end{array}$$

Having:

$$IDTS : TenantSet \to 1 \quad IDTS(x) = 1 \quad \forall x \in TenantSet,$$

$$IDUS : UserSet \to 1 \quad IDUS(x) = 1 \quad \forall x \in UserSet,$$

$$IDSR : STRING \times RealNumber \to 1 \quad IDSR(x) = 1 \quad \forall x \in STRING \times RealNumber,$$

$$IDR : RealNumber \to 1 \quad IDR(x) = 1 \quad \forall x \in RealNumber,$$

$$IDB : B \to 1 \quad IDB(x) = 1 \quad \forall x \in B,$$

$$IDPS : ProductSet \to 1 \quad IDPS(x) = 1 \quad \forall x \in ProductSet,$$

$$ATAT : UserSet \times TenantSet \to UserSet \quad ATAT(u,t) = concat(u, "@", t) \quad \forall (u,t) \in UserSet \times TenantSet,$$

$$buy : UserSet \times ProductSet \to UserSet \quad buy(u,p) = concat(u, "bought", p) \quad \forall (u,p) \in UserSet \times ProductSet,$$

$$exists : UserSet \to 1 \quad exists(x) = 1 \quad \forall x \in UserSet,$$

It is clear that when the number of places or transitions increases, the matrices become huge. Even though they are mostly zero matrices, the operation cost of real-world problems becomes complex.

The initial marking in MWS_CPN is

$$\begin{aligned} M_0 :=& (\{'e-commerce'\},\{\},\ \{'admin','user'\},\ \{'super'\},\ \{\}, \\ & \{\},\ \{\},\{('phone',250.0')\},\ \{1.0,-1.0\},\ \{\},\ \{\}) \end{aligned}$$

There are some relevant problems with this model, though. Supposedly, each tenant should be independent of the others. The product does not store information about the tenant. Any end user can buy all the tenant products! It is mandatory to keep track of the tenant associated with the data. Hereinafter, all the data related to a tenant will use the Cartesian product for its definition. The definition of all tenant related elements will have the structure

$$(tenantName, identifier) \qquad tenanName \in TenantSet,\ \ identifier \in \mathscr{C},$$

for proper visibility checks. An example of this Cartesian product is shown in Figure 43, see place P_U_S where the super user mark is a composition of its tenant and user name. The tenant linked to the super user is a meta-tenant allowed to handle the information of all the tenants.

Transitions can have restrictions based on the value of their input values. This possibility is equivalent to having a restriction on the input subset. In Figure 43 these issues are fixed:

- Transition T_O is not be enabled for products and users from different tenants.
- Only admins can create and modify products.

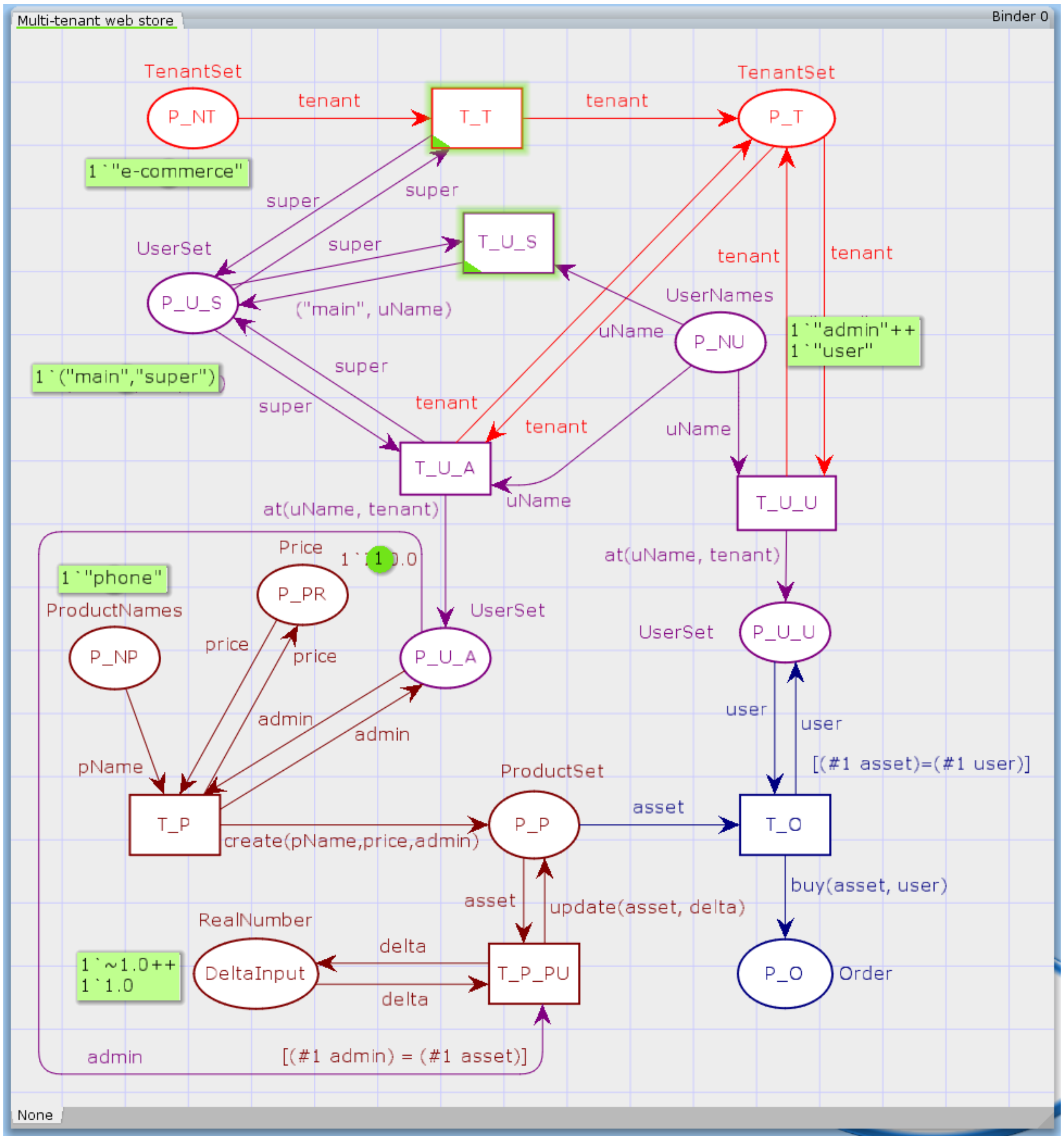


Figure 43: Multi-tenant Web Store secure model with a Colored Petri Net

The CPN in Figure 43 have several properties that will be discussed.
First, let's discuss security. The Net is K-limited with $K := \sup\{|cd(p)| : p \in P\} > 1$, verifying that the supremum is actually a maximum. Therefore, the net is not secure since it is not 1-limited.

Security is a very tough property not present in many nets modeling real-world systems considered safe. There is a trade-off between security and liveness. The decisions made in the design of the CPN prioritize more secure nets at the cost of liveness. The color domains are finite to represent the system database limitations at a given moment. If the system needs more resources, a color domain redefinition is required. It can be argued that the model should not be modified but represent all the theoretical resource needs. In that scenario, the net would improve its liveness properties at the cost of reduced K-limitation. The reachability net of the MWS CPN is finite.

To study if the net is conservative, we need to check if there is any transition generating infinite values or losing values. Most system transitions move marks from some places to others, without modifying the total number of marks. It is relevant to remember the distinction between conservative and strictly conservative net regarding the number of marks.

- T_T: the super user does not change, and the tenant mark moves to P_T.
- T_U_S: one mark moves from P_NU to P_U_S.
- T_U_A: one mark moves from P_NU to P_U_A.
- T_P: moves one product name mark to create a product.
- T_P_PU does not generate any mark. It just changes the product price.
- T_U_U: moves one mark from the user names to create a mark in users.
- T_O: moves one mark from the products to the orders.

It turns out that the number of marks is kept constant. The net is not just conservative, also strictly conservative and (see Figure 18). The net is not repetitive. It is possible to detect cycles in some subnets, but definitely not in the whole net. There is a clear path for the flow of data.

The data is created on demand, but the amount of tenants/users/products/orders is finite. If all the tenants create all the possible products, users, and orders, then there will be no possible transition to be fired. The trade-off here is K-limitation. We could sacrifice K-limitation to have infinite sets and have alive transitions. Therefore, the net is not alive, nor structurally alive. It acts as a complex producer-consumer model with limited resources. This limitation can be bypassed if necessary with color domain extensions.

About conflicts, transitions T_U_S, T_U_A, T_U_U are in an effective conflict. They all depend on a finite set of inputs to be enabled. The moment with just one mark in P_NU will have a conflict to choose the type of the last user to create. The concurrent nature of multi-tenancy approaches makes this kind of model inherently conflictive. In practice, since the tenants' data is logically separated at the application level, the conflicts are fewer than might appear at first glance.

The model does not show the specific database used. The restrictions of the same tenant for any product, user and order creation/modification represent the system's functional requirements. It is crucial to have tenants' data, at least, logically separated without direct access and/or modification from other tenants. The CPN model is database agnostic. It is independent of the usage of different databases, schemas or shared schemas. It just needs the data flow requirements. The order creation depends on the user (which is a functional requirement for purchases) and the product chosen. This information is convenient because it shows that the model is elastic until the product level. Elasticity at the product level means that, with vast sudden traffic demands, sharding data is allowed until product granularity in different databases. Therefore, it is possible to use dynamic scaling algorithms to split products into several databases on demand. The multi-tenant nature of the system makes dynamic scaling easier because it can be applied to all the tenants at the same time.

It is not feasible to handle the version of the reachability graph for this CPN (the occurrence graph) for all the scenarios. The complexity of generating the occurrence graph by hand is prohibitive. Only a computer can generate and analyze it with a certain guarantee of no errors. CPN Tools does not include tools to generate this graph Anyway, the relevant point is to understand the data flow in the model.

- First, any super user creates one or more tenants
- New users are created for the tenant (admins and end users)
- The admins of the tenant create/edit the products **only for its tenant**.
- The end users buy the products available in its tenant.

It turns out that there is only one possible data flow and a finite number of firings.

6 Quality of Service in Cloud Environments

In multi-tenant models, each tenant uses a fraction of the resources. Indeed, resource usage is very efficient when the tenant's load is not highly correlated. This tenant interdependency might hinder the quality of service defined in the Service-level agreement (SLA) with the client.

It is critical from the business perspective to ensure availability, performance, and elasticity of multi-tenant Cloud applications like MW-Store. The mean load might be relatively more stable when including more tenants. The problem is that the potential variance in traffic might affect response times if the system cannot scale adequately. Deficiencies in this model will affect all tenants simultaneously, breaking several SLAs. Any human intervention is slow and inefficient compared to a well-designed automated system. The principal scalability objective is to ensure dynamic adaptation to uneven loads without human intervention, verified with spike testing.

Enterprise software could potentially be used simultaneously by a lot of people. Performance is relevant in multi-tenant applications where a bottleneck affecting one customer could potentially affect several customers. We need to distinguish between data scaling (increased capability to store and handle data) and application scaling (total workload that the application can handle) [3].

Databases can scale up to increase server resources (memory, processors, disk speed, etc.) or scale out (a database partition into several servers) [3]. It is desirable to model the data clusters to optimize the read and write operations in a database. Dynamic scaling involves automatic detection and processing of resource usage metrics. Examples of metrics include CPU, memory, disk usage, etc. Based on the thresholds defined for these metrics, automatic triggers will perform scaling actions. It is also convenient to optimize resource usage by automatic descaling when needed.

6.1 Metrics

The minimum Quality of Service (QoS) without penalties is defined in a contract with each tenant called a Service-level agreement (SLA) [4]. Although the performance of the algorithms is relevant, database calls involving network incurs into a big percentage of the request latency. Adaptation to sudden load increases is critical to guarantee the restrictions of the SLAs [6].

By extension, response times must be under control in databases. This task is particularly difficult since the database handles all the concurrent data access abnormalities. QoS might involve response time limits. Dynamic bottleneck discover using spike tests helps to reduce maximum response times.

If not handled correctly, the system failures is the sum of all its components failures. Hardware failures are improbable, but possible. Its relevance could be reduced paying higher prices for Cloud services specialized in fault tolerance. Network failures are possible. Do not rely on network. There are different strategies to minimize the effect of network issues, e.g. network calls retry, idempotency, data replication, smart servers, Exactly-Once stream-processing (frameworks like Kafka allow this configuration by default), etc.

Some system properties must be verified to ensure QoS compliance:

- Elasticity: adaptation of the available resources to the necessary ones to handle the current system load without the stop-the-world problem.
- Availability: understood as the percentage of time when the system is reachable with valid responses. The interval of time without service between failures is reduced using self-healing and replication factor greater than one in container orchestration systems like Kubernetes. The application code scalability is simpler than data scalability due to the absence of state. When a program grows, it is desirable to split the code into different packages, creating independently deployable modules. A well-known availability measure is 9's. 3 nines uptime means that a system is fully operational 99.9% of the time [26].
- Load balance: it is desirable to have an evenly distributed load in the different system instances (application(s) and database(s)). Automatic load balancing provides maximum throughput with minimal response times [25]. Good load balancing might help achieve better elasticity (allowing more reaction time to adapt the system) and availability (reducing the risk of lack of resources. E.g. out of memory errors).

Another problem is the inter-dependency between tenants. A sudden load in a tenant could potentially affect all the tenants if the system is not elastic enough. In the real world, the data of one tenant is not usually directly accessible from another. The database must minimize tenant interference. It sounds straightforward, but the data store model must consider specific approaches to prevent this kind of interdependency. Techniques like data split, indexes, defined foreign keys, and so on minimize the data access interdependency.

6.2 Data scalability

Scalability must be taken into account in all parts of the system, including data stores. Otherwise, even if the application is extremely fast there will be a data access bottleneck. Application containers are stateless, making scalability a lot easier because it is possible to use load balance and automatic scaling approaches (i.e. using Kubernetes). With a certain amount of information or users, having the data only in one database is simply not feasible. If the application has only one data store, its resources are limited and also depend on a single-point-of-failure.

There are two approaches to achieving data scalability.

- Horizontal scalability (or scale-out): it includes new running instances to increase the number of possible handled requests. It is the most general and relevant technique. The downside of this approach in data stores is that it forces data migrations between data stores in critical scenarios.

- Vertical scalability (or scale-up): increase resources in a running instance, like a server. There are two possible implementations: partial redimension and total replacement.

There are two general horizontal data scalability patterns, replication, and partitioning.

6.2.1 Replication

Replication consists of creating different copies of a database (or database subset), ensuring their data integrity by synchronizing the replicas with the main database [3]. When write operations are only allowed to the original database, it is called single-master replication. When it is possible to write into several copies of the database is called multi-master replication and requires a more complex synchronization mechanism to integrate the changes between the different copies. It is a proper approach to select subsets of data in the database that are not modified frequently, and use single-master replication. Replication has the advantage of being potentially useful as a synchronized data backup. If the master database is down, it is possible to use the replica, preventing application outages. This increased security will only be successful if there is independent hardware in the master database and the replica database (i.e. different servers). Uncorrelated failures are extremely important. Otherwise, there would not be any security improvement.

There are several synchronization techniques. The synchronization techniques can be divided into two groups:

- Synchronous: Imagine that there is a change in the master data store. Then, before completing the update, the value in the other replicas will be invalidated/updated. There is no fraction of time when the data in the replicas is outdated. These techniques favour consistency at the cost of speed while updating. This kind of approach is desirable in scenarios where consistency must be kept at all costs, improving performance in other parts of the application.

- Asynchronous: Imagine the same change in the master data store. Then, the change will be applied to the master data store, even if that means there will be a fraction of time when data is outdated in the replicas. It favours speed over consistency, having eventual consistency because at some point the data in the replicas will be updated.

The data synchronization process is not trivial and still has several trade-offs for scalability and resilience. The ACID transactions become more complicated with distributed data. In some cases, a database transaction depends on the transaction(s) of another data store(s). A safe policy in a distributed transaction commit is to roll back by default. Only stop the rollback if a successful response is received. This approach enforces transitions resilient to partition tolerance and network issues. There are several protocols for distributed consensus in distributed data stores, like 2PC, 3PC or MVCC [24].

- 2 Phase Commit (2PC) [17]: in the first phase, a coordinator asks the different databases to create transactions in a prepared state, flushed into the disk. The database does not consider these rows on reads. If the whole transaction is later verified as finished, then update the row to a done state. The problem is that there are several data stores. It is theoretically possible to commit to one database and then have a network error with another. If the outage is solved, the data store that was down reviews the commit in a prepared state.
- 3 Phase Commit (3PC) [22]: evolution of 2PC. It attempts to solve the blocking problem if the coordinator is down. There is a new intermediate phase called pre-commit. The first phase is the same, ask the distributed databases to flush a transaction in a prepared state. If a predetermined number of data stores, k, respond to the initial request, a second message is sent to pre-commit the transaction. At this point, if the coordinator is down, a new coordinator is assigned. Thanks to the restriction of k responses, the new coordinator can continue the transaction if at least one of the data stores has the pre-commit state.
- Multi version concurrency control (MVCC) [24]: multiple versions of the same data are stored. PostgreSQL uses it to handle transactions. It is possible to choose dynamically between consistency or latency. In scenarios where eventually consistency is reasonable, this property removes locks between reads and writes. When strict consistency is required, the protocol returns the last version of each element with a possible cost of speed.

Now we will see the other data scalability pattern.

6.2.2 Partitioning

In partitioning, subsets of the database are moved to other locations/databases.

- In horizontal partitioning, a database table is divided into several databases maintaining its structure and reducing the number of rows handled by each database. This is also called sharding and require a mechanism to join the resources together, and ensure correctness of the queries. Load balancing redirects traffic to the required sharded database(s). In order to distribute the rows of the database in several ones a distribution algorithm is needed. That algorithm must optimize the even distribution of data, like the module operator. The problem with the module operator is that it is not optimal to dynamically add or remove databases for dynamic horizontal scaling, since the ratio of data to move would be too high. Other techniques like consistent hashing with virtual nodes decrease data redistribution when adding/removing database servers on demand.
- In vertical partitioning, the columns of a table are split into several tables that will have the same number of rows that the original table. This technique is useful when a monolithic application is split into several modules. Splitting the tables in a schema into several subsets allows for independently modules deploy. As a rule of thumb, database divisions should minimize the number of cross-database communications, keeping related tables in the same database.

It is possible to create horizontal partitioning in a shared database based on tenant ids. Different tenants can have very different demands. It is necessary to prevent overtaxed partitions while other partitions are underused, i.e. partition the database to balance the number of requests on each database.

Using partitioning also reduces the single-point-of-failure issue. If data is properly distributed (related data kept together) then one part of the distributed data store can be down but most of the requests can still succeed using the other data stores. Partitioning can be used combined with replication for increased security and availability, at the cost of complexity and potential eventual consistency.

6.3 Database types

It is relevant to study how different databases behave under load. The concurrent nature of multi-tenant applications makes this study especially important. Different databases have distinct goals, use cases and trade-offs. There are two main approaches to fetch data:

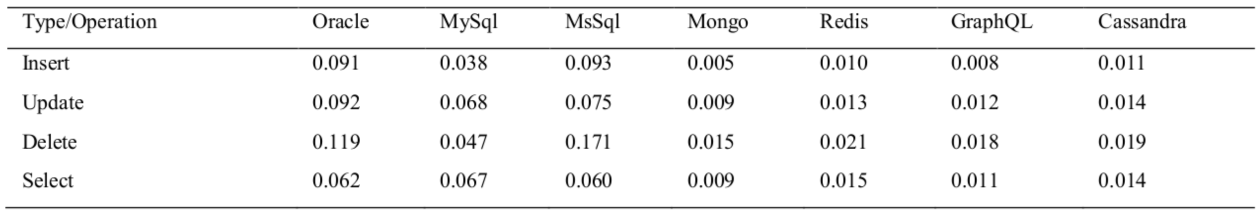
- SQL (Structured Query Language): this query language to interact with data is typically linked to relational databases. These relational databases store data in tables and allow checking the foreign table relationships at the database level. MySQL or PostgreSQL are examples of relational databases.
- NoSQL (Not-Only relational or Non-relational): it does not define the relationships, allowing extra flexibility but delegates the checks of the possible hidden relationships in the data. There is no standard query language for NoSQL. Each NoSQL database defines its query language, adapted to the topology of the database. UnQL (Unstructured Query Language) is a super set of SQL attempt to standardize a query language for NoSQL databases. UnQL allows querying data in JSON or document formats. Redis or MongoDB are examples of non relational databases.

The relational model is arguably more structured but it has some scalability problems with high volumes of data [18]. NoSQL database systems relegate consistency management to create a more scalable data storage. Inside NoSQL databases there are several types [18]:

- Key-Value Store Databases: the key and the value are string. Typically, the key is a simple type, and the value can represent a serializable object of a programming language. Examples are Redis or Amazon DynamoDB.
- Column-Oriented Databases: each item has a key related to one or more column(s). The NoSQL database more similar to SQL ones, but the internal storage method is distributed and optimizes column operations. An example of this type of database is Cassandra.

- Document Store Databases: data is stored as documents using a key. Documents are flexible records without a fixed schema, allowing formats like XML, PDF or JSON. These databases are more complex than key-data ones but allow more granular searches filtering by document fields, reducing network latency. Examples are MongoDB or CouchDB.
- Graph Databases: data is stored as objects (nodes) and their relationships (edges). These edges are pointers to the adjacent nodes. It is considered semi-structured data, there are relationships between the nodes, and each node is schema-less. Rollbacks are supported as well as the ACID properties. An example of a native graph database is Neo4j.
- Object Oriented Databases: data is stored as objects and offers object-oriented features like data encapsulation, polymorphism and inheritance. Objects can are referenced with pointers for faster access.

Not all the databases are directly comparable. Redis, for example, is an in-memory cache and compared SQL databases are not (although they might use some kind of in-memory cache to improve performance). The performance differences are shown in Figures 44, 45 and 46. It turns out that there are relevant differences between SQL and NoSQL databases that deserve some comments.



Type/Operation	Oracle	MySql	MsSql	Mongo	Redis	GraphQL	Cassandra
Insert	0.091	0.038	0.093	0.005	0.010	0.008	0.011
Update	0.092	0.068	0.075	0.009	0.013	0.012	0.014
Delete	0.119	0.047	0.171	0.015	0.021	0.018	0.019
Select	0.062	0.067	0.060	0.009	0.015	0.011	0.014

Figure 44: Query performance of database with 100 000 records in milliseconds [2].

The numbers show a clear advantage of NoSQL vs SQL databases. They are different databases, created for different purposes. Redis is an in-memory cache, but it is surprising the magnitudes of MongoDB vs relational databases. The results of the CREATE/UPDATE/READ/DELETE are combined for approximate overall comparisons in Figures 45 and 46.

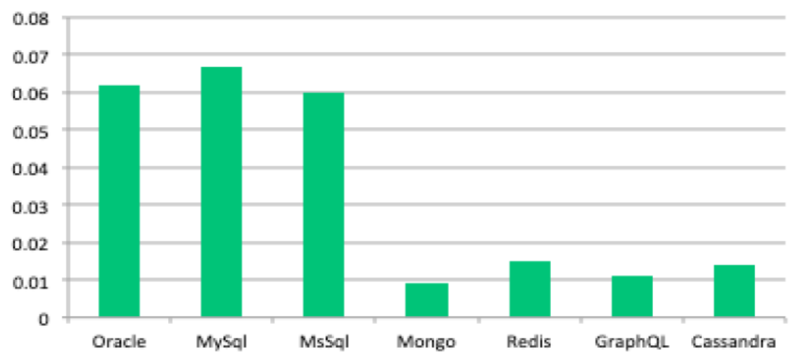


Figure 45: Query performance in milliseconds [2].

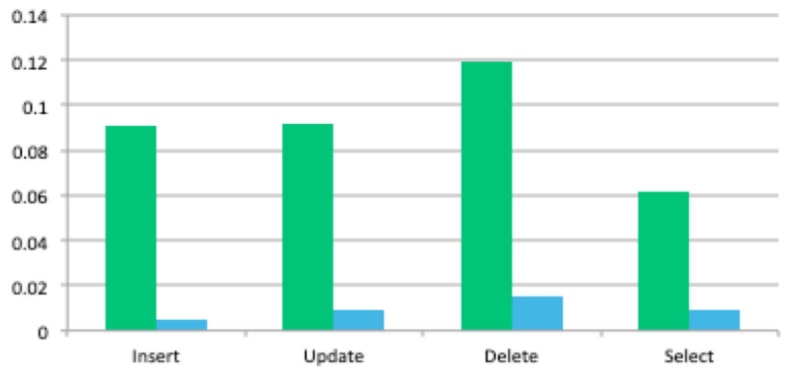
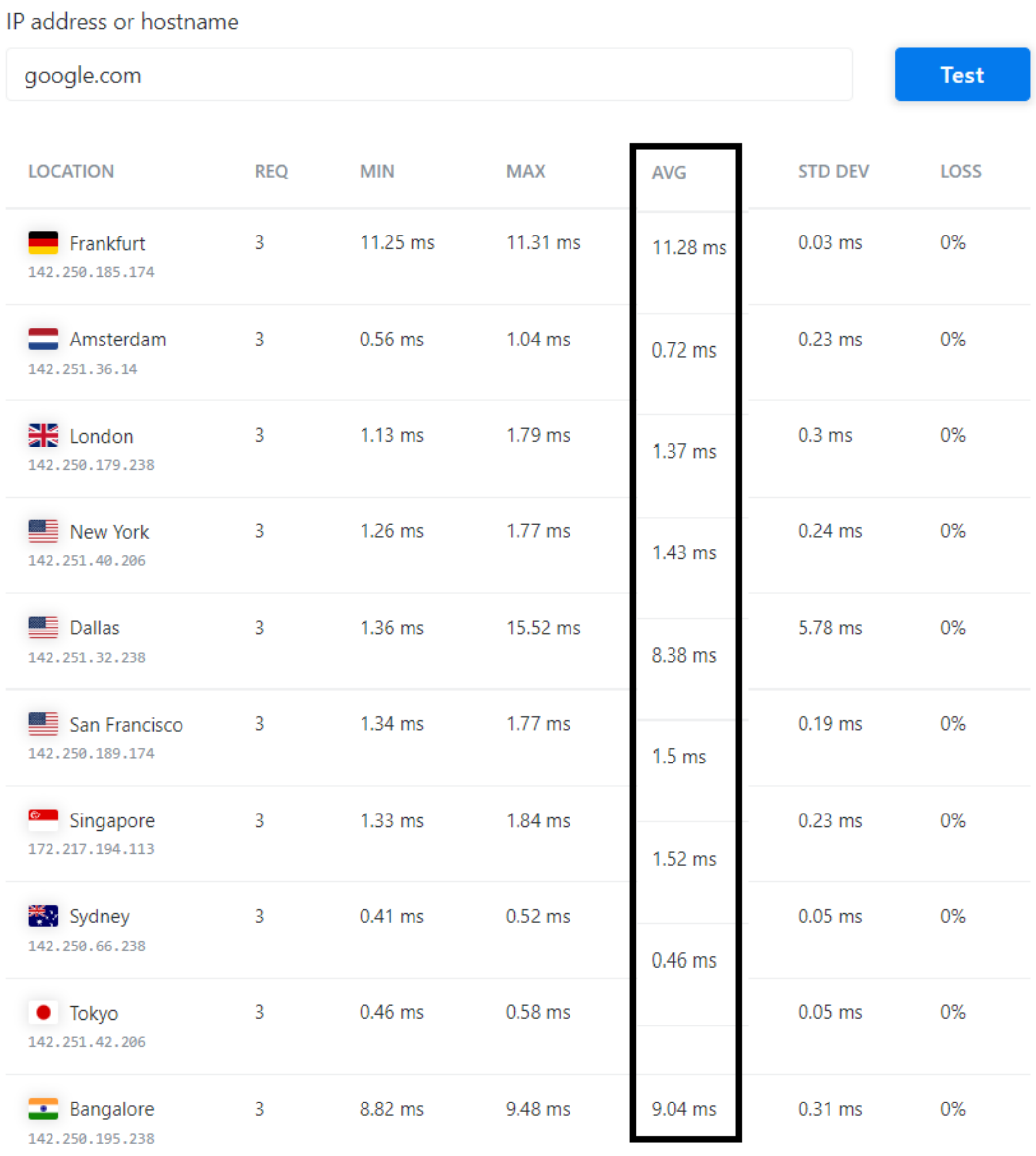


Figure 46: Query performance for Oracle (green) and MongoDB (blue) [2].

NoSQL databases store data in compact entities, allowing fast manipulation. SQL databases encourage normalization, reducing redundancies and undesirable consistency issues by dividing tables into smaller ones connected through relationships. These relationships enforce a structure in the data model but prevent the database to reach the performance of NoSQL.

The performance tests must include all the intermediate steps of a process. Figure 47 compares the magnitude of ping response times worldwide. It turns out that for relatively small amounts of data, the performance differences are not a big deal [2]. The problem might arise with extensive amounts of data and complex data relationships enlarging the performance difference due to internal data representation differences between SQL and NoSQL databases.



IP address or hostname

google.com

Test

LOCATION	REQ	MIN	MAX	AVG	STD DEV	LOSS
Frankfurt 142.250.185.174	3	11.25 ms	11.31 ms	11.28 ms	0.03 ms	0%
Amsterdam 142.251.36.14	3	0.56 ms	1.04 ms	0.72 ms	0.23 ms	0%
London 142.250.179.238	3	1.13 ms	1.79 ms	1.37 ms	0.3 ms	0%
New York 142.251.40.206	3	1.26 ms	1.77 ms	1.43 ms	0.24 ms	0%
Dallas 142.251.32.238	3	1.36 ms	15.52 ms	8.38 ms	5.78 ms	0%
San Francisco 142.250.189.174	3	1.34 ms	1.77 ms	1.5 ms	0.19 ms	0%
Singapore 172.217.194.113	3	1.33 ms	1.84 ms	1.52 ms	0.23 ms	0%
Sydney 142.250.66.238	3	0.41 ms	0.52 ms	0.46 ms	0.05 ms	0%
Tokyo 142.251.42.206	3	0.46 ms	0.58 ms		0.05 ms	0%
Bangalore 142.250.195.238	3	8.82 ms	9.48 ms	9.04 ms	0.31 ms	0%

Figure 47: Response times for google.com in ten different locations. Tool: https://tools.keycdn.com/ping).

At this point, there is a trade-off between performance (or potential scalability) and flexibility against structure, reduced duplicity and consistency guarantees. It can be argued that flexibility is not a massive advantage. In the multi-tenancy customization section, several extensibility patterns of the data model were proposed. It was pointed out that some databases, like PostgreSQL allow JSON column types with custom queries.

Consistency trade-offs will be discussed in the next section, comparing the different database transaction models. Then, performance will be compared, for multi-tenancy scenarios, in the section of 'Quality of Service in Cloud Computing'.

6.4 Comparison of Database Transaction Models

There are some relevant differences between the different databases types, like transaction models. A transaction is a set of instruction(s) considered a single unit, handling system's state. In SQL databases, a transaction begins, then it succeeds (the transaction is committed and persisted to the database) or fails (some databases support rollbacks command to prevent any change in the database if a transaction fails). Sometimes a transaction is divided into several transaction units. If any of these unit fails, the whole transaction is marked as a failure. Most relational database systems guarantee the ACID properties [15]: Atomicity, Consistency, Isolation and Durability.

- Atomicity: All the tasks inside a transaction are applied, or none. One task failing implies that the whole transaction fails and nothing is committed. This property prevents partial writes into the database.
- Consistency: The transactions always verify the constraints of the system. They can not be committed leaving an inconsistent state in the database.
- Isolation: Every transaction is independent and has no access to the information of other unfinished transactions.
- Durability: After transaction persistence, it is not possible to revert its effects, and they will be resilient to breakdowns in the system.

These ACID properties are a safety net at the database level, allowing to have reliable data storage. After a certain amount of load, the need for data store scaling created new challenges and encouraged the review of the ACID restrictions. A relational database can still be horizontally scalable using distributed databases. But, due to its table structure, the performance of the queries depends on the number of tables involved. Handling data in a tree structure(s) allows the database to store the data physically closer and makes distributed databases easier to scale by design.

It is possible to fetch data using the root of the three as id. This possibility to fetch all the data together allows for improved performance. A potential problem in distributed databases is queries that force a full scan of all the databases. If the application continuously scans all the databases filtering for a field, then the advantages of this distribution are under threat.

Storing data in a tree structure(s) is more flexible than table structure(s). As we saw previously, data customizability patterns can be defined in relational models, even though some of them could act as a black box. Some relational databases (like PostgreSQL) allow having column(s) with JSON format (with the possibility to query inside the JSON fields), having the advantages of the relational model and the flexibility of the tree structure.

On the other hand, NoSQL databases focus on performance, scalability, and flexibility. The origin of the popularity of the term BASE is thanks to Dr Eric Brewer in the Symposium "Towards Robust Distributed Systems" [15] NoSQL does not support transactions [2] and will not always guarantee ACID properties, having softer constraints: Basically Available, Soft state and Eventual consistency (BASE) [15].

- Basically Available: There is a certain degree of availability even with some node unavailability. The trade-off here is to allow data to be outdated for the benefit of increased availability.
- Soft state: Data consistency is not always guaranteed. The returned data could not be the latest state.
- Eventual consistency: There is a mechanism to update the data in all the data storage. It is guaranteed to have a consistent state eventually.

The BASE paradigm can be used as relaxed constraints when approximate answers are tolerable. ACID paradigm is more suitable for conservative scenarios where data must guarantee to be consistently updated, sacrificing availability. When the availability is sacrificed, then it is possible to send multiple requests, but returned data is guaranteed to be fresh. If consistency is not a must, it is possible to create mechanisms to minimize this issue. For example, in data updates, before updating the master data, mark the data in the nodes as outdated. ACID is better suited for applications that are constantly inserting and updating data and want this layer of security. BASE is better for applications that needs extreme performance in read-only operations.

Another possibility is to combine both systems. It is possible to have the core system verifying ACID for insert(s)/update(s) and data replications with BASE properties to have better scalability. It turns out that the ACID vs BASE debate is a spectrum that depends on the specific system requirements. This trade-off is described for any distributed data storage.

6.5 Distributed data store trade-offs

When an application needs to access data, there is a layer between that application and the data store. This layer could be a network call between two servers or a process inside the same server. Hereafter, it will be referred to as a distributed data store because it can be generalized to any data allocation between one or multiple servers. What happens if the link with the data store breaks? Are there any potential issues while writing and reading data concurrently? The formalism to properly define these concepts will be shown. Then, a discussion about their limitations and possible trade-offs depending on the system requirements.

It turns out that in distributed data stores it is not possible to have complete availability and data consistency simultaneously. That limitation is formally stated in the CAP theorem. The original article proving the CAP theorem named it the ”Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services" [7]. It references a limitation of distributed web services viewed as asynchronous network models.

Theorem 1 (**Brewer's Conjecture (CAP)**)
It is impossible for a web service to provide the following three guarantees:

- Consistency
- Availability
- Partition-tolerance

The Brewer's Conjecture presented lacks concise definitions and proof. The formal demonstration requires several definitions not included in this project but can be found in [7]. It is considered one of the precedents in the debate about the limitations in the design of ACID databases. The traditional view of interaction with the database behaves as a transaction. An operation is atomic when it is completed (committed) or fails without intermediate states. Consistency is the impossibility of inconsistent data at any time that does not verify the rules defined in the database. Isolation means that a transaction does not have information about other transactions. It is possible to detect overlapping writes that could lead to inconsistent data and verify transaction isolation since the transaction could have information about the previous system state. The previous system state does not necessarily mean having information about other transactions. Several transactions can finish before our operation without having information about them. The only knowledge is that there was a data change. Another property is durability, understood as the impossibility of rolling back a change when complete. Sensitive critical information should still have this strong consistency because it is the final security layer of a critical system.

Consistency is a very general term. There are more specific definitions of consistency, with different properties, granularity, etc. Some of these consistency types are [15]:

- Strict Consistency: Every read provides the last updated value. If there are data replicas, all must return the same value after any write operation in one of them.
- Eventual Consistency: reads could return outdated value. The only guarantee is that, at some point, all replicas will return the same value.
- Monotonic Read Consistency: reads could return outdated value. It guarantees that if there are several reads over the same object, the client will receive either a previous value or a more updated value.
- Read Your Own Writes: when there is an update (independently of the replica to write), the client always receives that updated value, no matter the requested replica node.
- Causal Consistency: causally related writes must be seen by all processes in their written order.

- Linearizability: It is a way to describe concurrent operations as a sequence of non-blocking instantaneous events verifying the First In First Out (FIFO) property. A system is linearizable when all its objects are. A formal definition with writes and reads operations is described in [10]. The intuitive consequences are that after an instant modification of a resource, any posterior read operation will see that value if there is no other modification (see Figures 48 and 49).

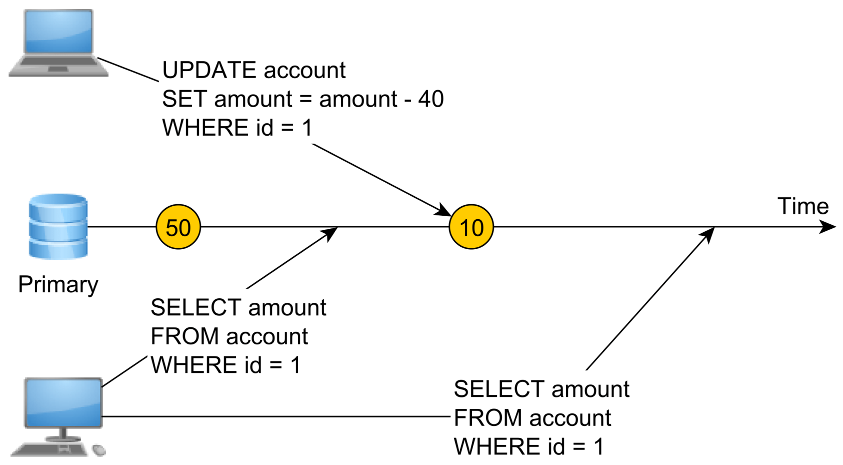


Figure 48: Linearizable operations [16]. Read operations always return the latest write.

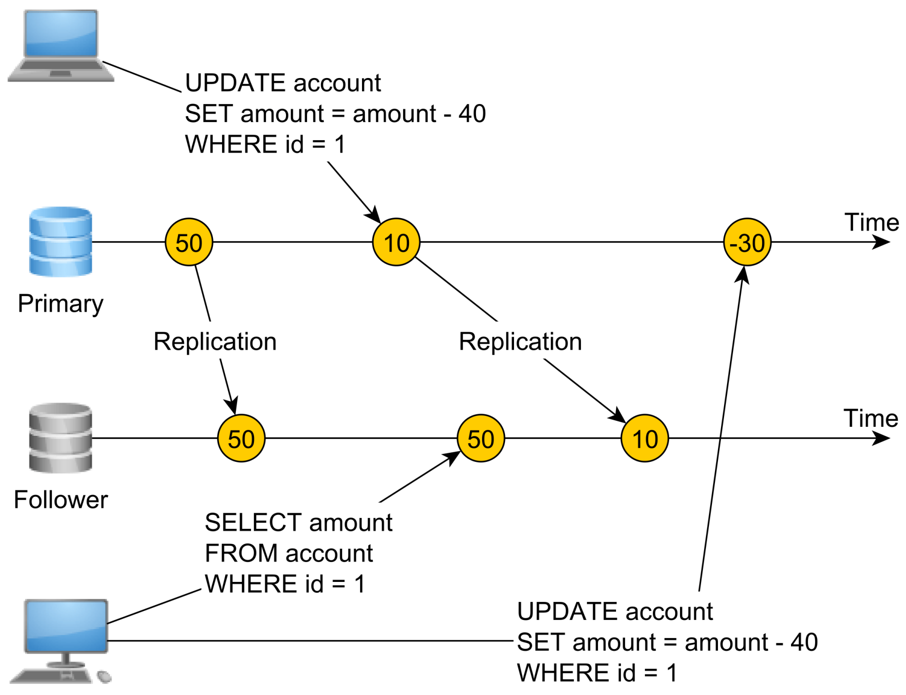


Figure 49: No linearizable operations in a data store with a replica [16]. A node reads a value in the replica before the previous write operation sync and then tries to write in the primary data store. The system is not linearizable since the data replication is asynchronous.

There are other examples of consistency guarantees, i.e. FIFO, sequential, entry, casual+, consistent prefix, release, or bounded staleness. It is clear now that the different consistency definitions might impact the other two properties (Availability and Partition Tolerance) of the CAP theorem. That is the motivation for the *PACELC theorem* to increase the granularity of the consistency definition.

The *PACELC theorem* suggests that, in a distributed system, with a partition (P), it is only possible to choose between (A) or (C) (Availability or Consistency), else (E), without partitions, there is still a trade-off between (L) and (C) (Latency or Consistency) [8].

It applies softer constraints to the scenario of just one partition, allowing the possibility to change availability with just latency. The possible combinations of distributed replicated systems: PC/EC, PC/EL, PA/EL, and PA/EC. Other authors consider that the relationship between availability and consistency should be a continuum, from weak to strong, with eventual consistency in the middle [8].

7 Conclusions

Multi-tenancy is a convenient paradigm with several advantages like cost savings and easier client extensibility. There are other disadvantages, like the interdependency created between tenants. Huge loads in one tenant might reduce the Quality of Service for all the tenants. This interdependency creates strong incentives to consider scalability as a dynamic property called elasticity.
Petri Nets allow us to model any program. The implications of this modelling are huge. Some properties deducible from CPNs are resource availability, system liveness, security and elasticity to make the system more robust under extreme workloads.
In conclusion, being aware of these concepts help to design better software architectures. This project shows another example of how the combination of computer sciences and mathematics helps to solve real-world problems.

References

[1] Michel Beaudouin-Lafon et al. “CPN/Tools: A Post-WIMP Interface for Editing and Simulating Coloured Petri Nets”. In: *Laboratoire de Recherche en Informatique* (2001). URL: https://www.researchgate.net/publication/317219197_CPNtools_A_post-WIMP_interface_for_editing_and_simulating_coloured_petri_nets.

[2] Roman Čerešňák and Michal Kvet. “Comparison of query performance in relational a non-relation databases”. In: *Transportation Research Procedia* (2019).

[3] Frederick Chong, Gianpaolo Carraro, and Roger Wolter. “Multi-Tenant Data Architecture”. In: *Microsoft Corporation* (2006).

[4] Brian Cooper et al. “Building a Cloud for Yahoo!” In: *IEEE Data Engineering Bulletin, 32(1), pp.:36-43* (2009).

[5] Manoj Debnath. “Understanding Views in SQL”. In: *Database journal* (2020).

[6] Aaron Elmore et al. “Zephyr: live migration in shared nothing databases for elastic Cloud platforms”. In: *SIGMOD-2011, pp.:301-312* (2011).

[7] Seth Gilbert and Nancy Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. In: *ACM SIGACT News* (2002), pp. 51–59.

[8] Anatoliy Gorbenko, Andrii Karpenko, and Olga Tarasyuk. “Analysis of Trade-offs in Fault-Tolerant Distributed Computing and Replicated Databases”. In: *2020 IEEE 11th International Conference on Dependable Systems, Services and Technologies (DESSERT)*. 2020, pp. 1–6. DOI: 10.1109/DESSERT50317.2020.9125078. URL: https://ieeexplore.ieee.org/abstract/document/9125078.

[9] Ville Hartikainen. “Defining suitable testing levels, methods and practices for an agile web application project”. In: *LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT* (2020), pp. 18–19.

[10] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: A Correctness Condition for Concurrent Objects”. In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. URL: https://doi.org/10.1145/78969.78972.

[11] Hector Humanes et al. “Estudio del Soporte a la Variabilidad en la Cloud en un entorno con Multitenencia: Plataforma GPaaS”. In: *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)* (2016).

[12] Jaap Kabbedijk et al. “Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective”. In: *The Journal of Systems and Software* (2014).

[13] Seungseok Kang et al. “A General Maturity Model and Reference Architecture for SaaS Service”. In: vol. 5982. Apr. 2010, pp. 337–346. ISBN: 978-3-642-12097-8. DOI: 10.1007/978-3-642-12098-5_28.

[14] Allan Kelly. “The Philosophy of Extensible Software”. In: *ACCU* (2002).

[15] Mr. Keith Machado et al. “A Comparative Study of ACID and BASE in Database Transaction Processing”. In: *International Journal of Scientific & Engineering Research* (2017).

[16] Vlad Mihalcea. *What is "Linearizability"?* 2012. URL: https://stackoverflow.com/questions/9762101/what-is-linearizability.

[17] Hussein Nasser. *Distributed Transactions are Hard (How Two-Phase Commit works)*. 2022. URL: https://www.youtube.com/watch?v=eltn4x788UM&ab_channel=HusseinNasser.

[18] Ameya Nayak, Anil Poriya, and Dikshay Poojary. “Type of NOSQL Databases and its Comparison with Relational Databases”. In: (2013).

[19] Amir Ngah, Malcolm Munro, and Mohammad Abdallah. “An Overview of Regression Testing”. In: *Journal of Telecommunication, Electronic and Computer Engineering* (2017).

[20] Max Rehkopf. “What is continuous integration?” In: *Atlassian documentation* (2022).

[21] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. “Regression test selection for C++ software”. In: *Journal of Software: Testing, Verification and Reliability* (2000), pp. 77–109.

[22] Rupali. *Three Phase Commit Protocol*. 2022. URL: https://www.youtube.com/watch?v=o0OAcd5B23c&ab_channel=Rupalima%27am%27sclass.

[23] Oliver Schiller et al. “Native support of multi-tenancy in rdbms for software as a service”. In: *In Proceedings of the 14th International Conference on Extending Database Technology, EDBT/ICDT ’11, pp.:117-128* (2011).

[24] Gaurav Sen. *Distributed Consensus and Data Replication strategies on the server*. 2019. URL: https://www.youtube.com/watch?v=GeGxgmPTe4c&ab_channel=GauravSen.

[25] R. Shimonski. “Windows 2000 and Windows Server 2003 Clustering and Load Balancing”. In: (2003).

[26] stratus.com. *Server Uptime Meter*. 2022. URL: https://www.stratus.com/about/company-information/uptime-meter/#:~:text=Availability%20is%20normally%20expressed%20in,have%20on%20your%20server%20downtime.

[27] szaidigmail. *Unit 1-Video 1-Introduction to CPNTools Interface*. 2012. URL: https://www.youtube.com/watch?v=38g1jMvNi6Q&ab_channel=szaidigmail.

[28] CPN Tools. *Arc inscriptions*. 2018. URL: http://cpntools.org/2018/01/09/arc-inscriptions/.

[29] Manuel Isidoro Capel Tuñón. *Lecture notes for subject ’Programación Concurrente’, Tema 6: Modelización de Sistemas Concurrentes*. 2015. URL: https://lsi2.ugr.es/~pc/teoria.htm.

[30] *Types Of Software Testing: Different Testing Types With Details*. URL: https://www.softwaretestingh
com/types-of-software-testing/#1_Security_Testing (visited on 05/05/2022).

[31] Lisa Wells. “Performance analysis using CPN tools”. In: *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools, VALUETOOLS 2006, Pisa, Italy, October 11-13, 2006*. Ed. by Luciano Lenzini and Rene L. Cruz. Vol. 180. ACM International Conference Proceeding Series. ACM, 2006, p. 59. DOI: `10.1145/1190095.1190171`. URL: `https://doi.org/10.1145/1190095.1190171`.

[32] Qi Zhang, Lu Cheng, and Raouf Boutaba. “Cloud computing: state-of-the-art and research challenges”. In: *Journal of Internet Services and Applications* (2010), pp. 7–18.