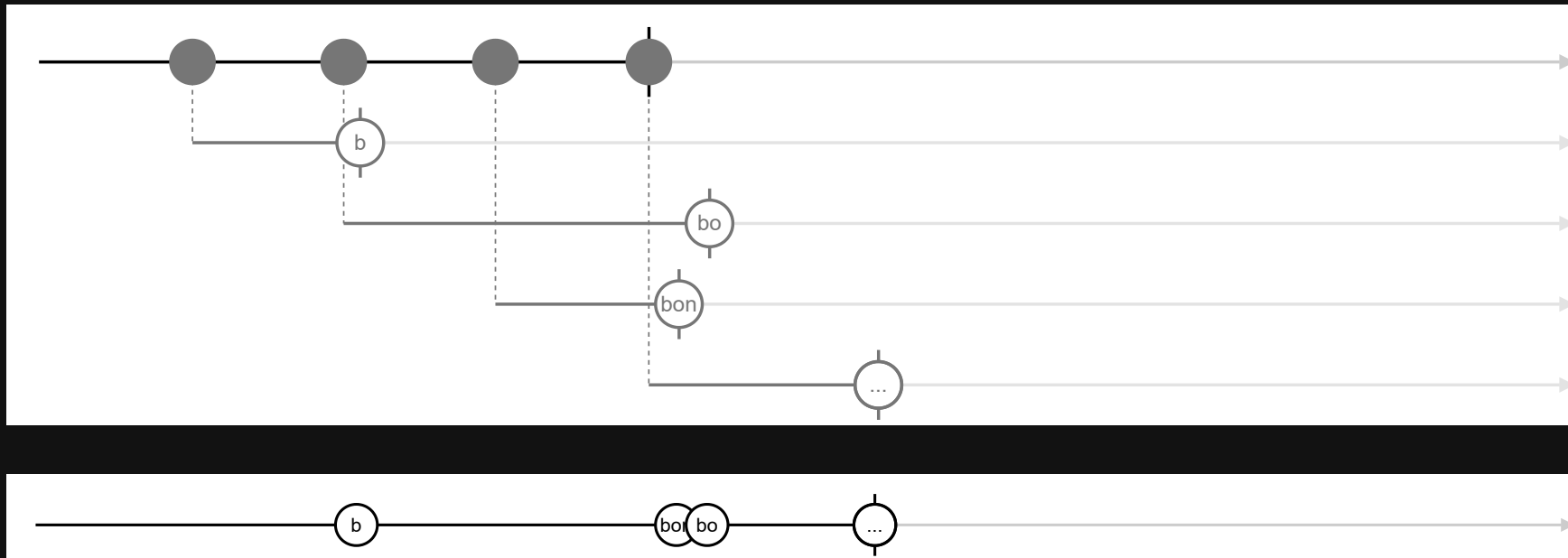
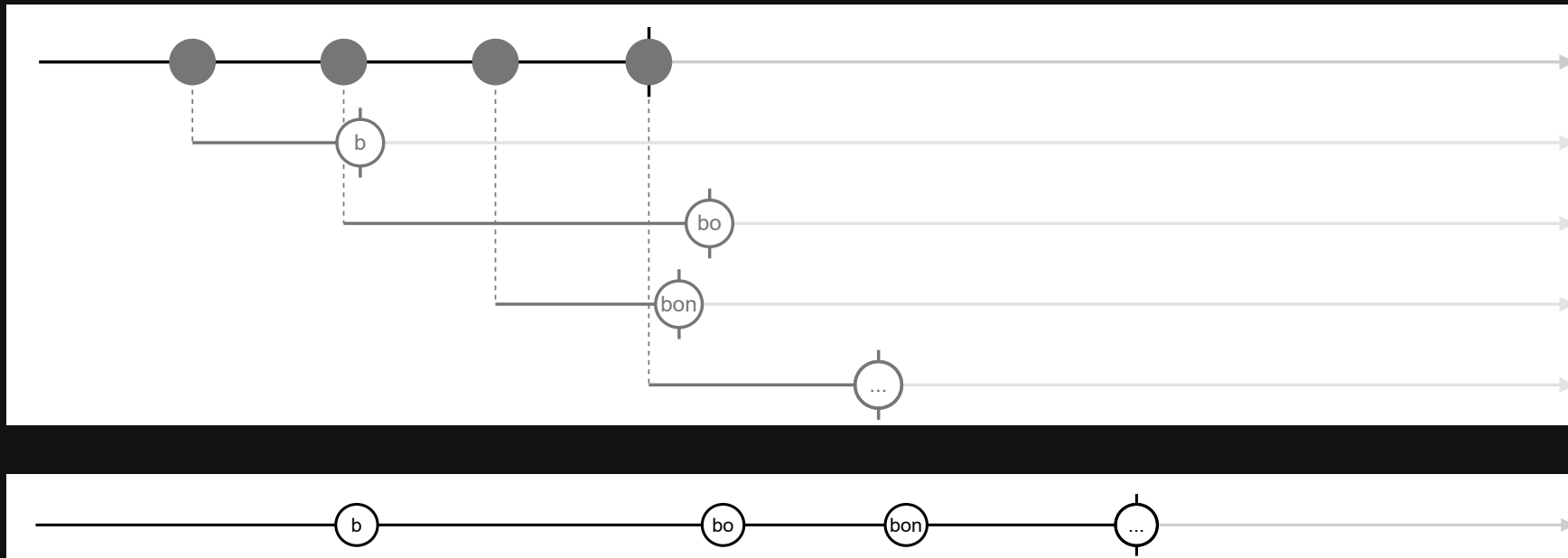


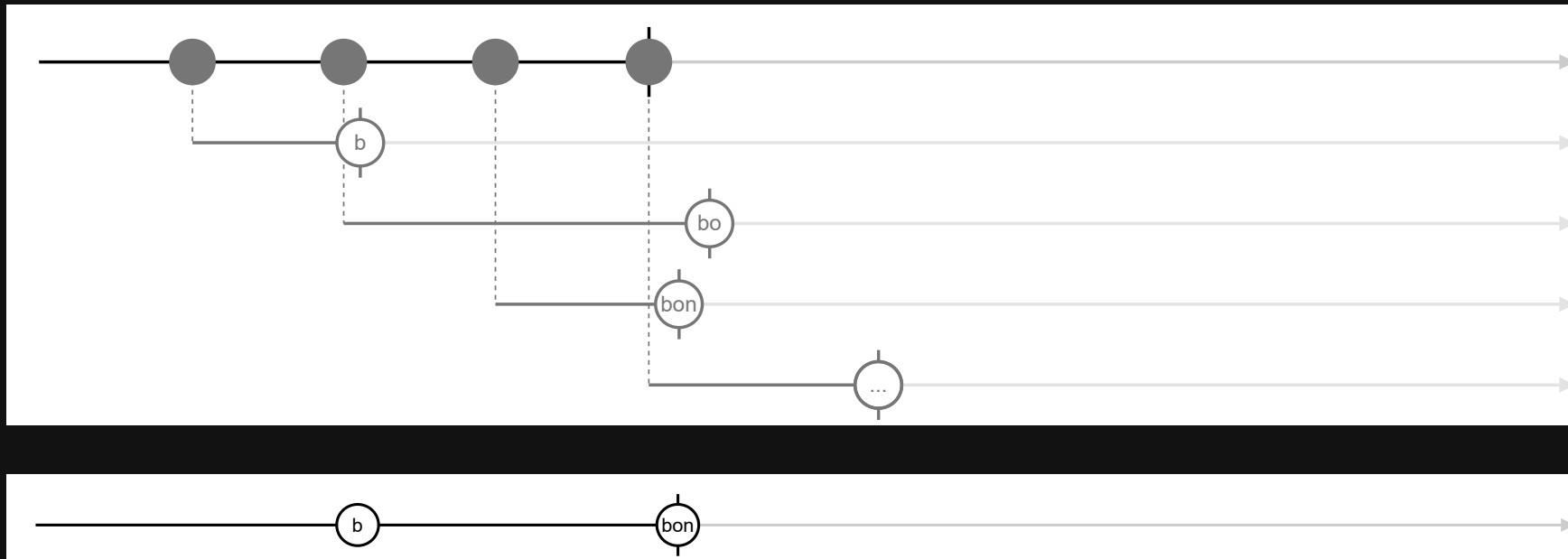
Avec mergeMap()



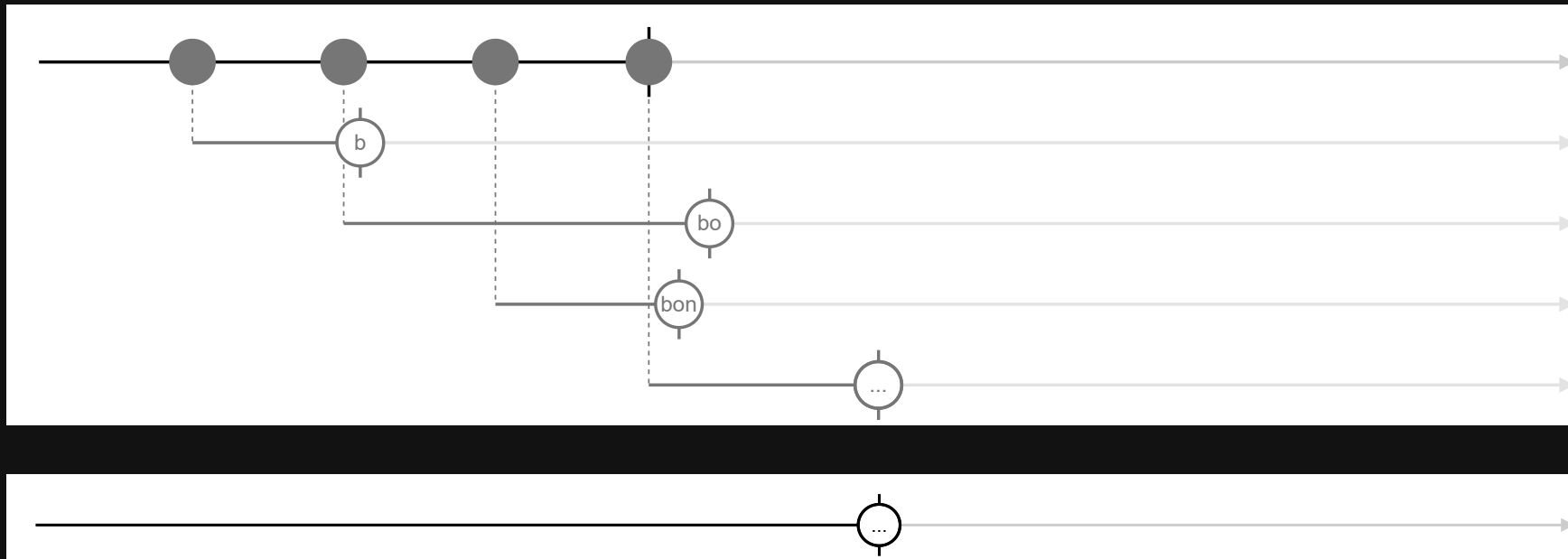
Avec concatMap()



Avec exhaustMap()



Avec switchMap()



Observable et stratégie OnPush

- Au sein d'un composant avec la stratégie OnPush, que se passe-t-il lorsque l'un de ses observables émet une nouvelle valeur ?
- Rappel des conditions pour mettre un composant dirty :
 - Un événement du DOM lié à ce composant a lieu
 - Une variable @Input ou @Output de ce composant change
- Aucune des deux conditions n'est remplie pour mettre le composant dirty, l'affichage n'est donc pas mis à jour
- ⚠ Bien que le composant ne se mette pas à jour, l'observable émet ses valeurs, et les cycles de détection de changement ont lieu
- Et avec le pipe async ?
 - Le pipe async met automatiquement le composant dirty

State managment

State managment

- Retour sur le service CounterService
- Notre service gère un état (la valeur du compteur) qui peut être partagé par plusieurs consumers
- Pour le moment, cette gestion n'est pas faite de manière réactive
- Que se passerait-il si on veut utiliser notre service avec des composants OnPush ?
 - Les composants ne sont pas marqués dirty
- Nous allons voir comment utiliser RxJS pour gérer les états de notre application de manière réactive

Subject

- Un Subject est un type particulier d'Observable

```
export class Subject<T> extends Observable<T> ...
```

- Comme un observable, on peut définir des observateurs qui s'abonnent au subject

```
subject = new Subject()  
  
subject.subscribe(observer1)  
subject.subscribe(observer2)
```

- A la différence des Observables, les valeurs émises par le Subject sont multicast
- Dans l'exemple précédent, cela implique que l'on aura une seule execution du subject pour les deux observateurs (à la différence d'un observable classique, où chaque observateur aurait eu sa propre execution)

Subject

- La classe Subject implémente les méthodes `next(v)`, `error(e)` et `complete()`

```
subject.next(v)  
subject.error(e)  
subject.complete()
```

- Lorsque on appelle `next(v)`, le subject émet une valeur à ses observateurs
- Lorsque on appelle `complete()`, le subject complète
- Lorsque on appelle `error(v)`, le subject envoie une erreur

Subject

- Le nom des méthodes `next()`, `error()` et `complete()` n'a pas été choisi par hasard
- Comme il implémente ces méthodes, notre subject est également un observateur
- Comme un observateur, notre subject peut s'abonner à un observable :

```
observable$ = new Observable(...)  
  
observable$.subscribe(subject)
```

- Dans ce cas, le subject est utilisé comme intermédiaire pour émettre les valeurs d'un observable en multicast à plusieurs observateurs

BehaviorSubject

- Le BehaviorSubject est un type particulier de Subject

```
class BehaviorSubject<T> extends Subject<T> {
```

- Le BehaviorSubject à une notion de valeur courante (la dernière valeur émise), qu'il est possible de récupérer

```
behaviorSubject.getValue()
```

- Lorsque un observateur s'abonne au BehaviorSubject, il reçoit immédiatement la valeur courante
- Un BehaviorSubject doit toujours avoir une valeur courante, pour cela il est nécessaire d'en donner une par défaut lorsque on instancie un BehaviorSubject

```
behaviorSubject = new BehaviorSubject(0)
```

State management avec RxJS

- Nous allons voir une manière d'implémenter le state management avec RxJS, en refactorisant notre CounterComposant de manière réactive :
- La base de notre state management va être un BehaviorSubject

```
private _state = new BehaviorSubject(0)
```

- On ne veut en général pas exposer directement le BehaviorSubject dans notre service, pour limiter ce que peuvent faire les utilisateurs du service, pour cela on peut convertir un Subject en Observable

```
observable$ = this._state.asObservable()
```

State management avec RxJS

- Les consumers peuvent s'abonner à l'observable pour récupérer l'état du compteur

```
reactiveCounterService = inject(ReactiveCounterService)
```

```
{{ reactiveCounterService.counter$ | async }}
```

- Et l'incrémenter avec une méthode du service

```
incrementValue() {  
  this._counter.next(this._counter.getValue() + 1)  
}
```

State management et objets

- Très souvent, on va vouloir représenter notre état sous la forme d'un objet
- Comme d'habitude, il y a une différence de comportement entre un objet (référence) et un type primitif (valeur)
- Il est très important que les abonnés à l'observable ne modifient pas directement l'état, car l'état serait modifié pour les autres abonnés sans qu'il y ait de notification

Exercice - Application magasin

- Implémenter un service "panier" avec RxJS
 - L'état du panier est stocké dans un BehaviorSubject
 - L'utilisateur peut ajouter ou enlever un objet dans le panier, et vider le panier
- Implémenter le service "produits"
 - Une méthode permet de récupérer la liste des produits disponibles depuis un serveur distant