React 19 Concurrency: What, Why, and How

React Paris Meetup #011

2025

Contents

React Paris Meetup #011

Tonight's Talks

Talk 1: Conquering React Concurrency

Ariel Shulman

Deep dive into React's concurrent rendering model and practical patterns.

Talk 2: Evolution des librairies UI et futures problematiques

Theo Senoussaoui

The evolution of UI libraries and future challenges in the React ecosystem.

Companion Materials

This presentation provides supplementary notes covering:

- Concurrency fundamentals
- React 16-19 evolution
- ClojureScript comparisons (Reagent, re-frame, core.async)
- 10 practice projects

Repository: `https://github.com/jwalsh/react-paris-meetup-011`

Concurrency vs. Parallelism

The Classic Distinction

Concurrency

- **Dealing** with multiple things at once
- Single core, interleaved execution
- About **structure**
- "Managing multiple tasks"

Parallelism

- **Doing** multiple things at once
- Multiple cores, simultaneous execution
- About **execution**
- "Running multiple tasks"

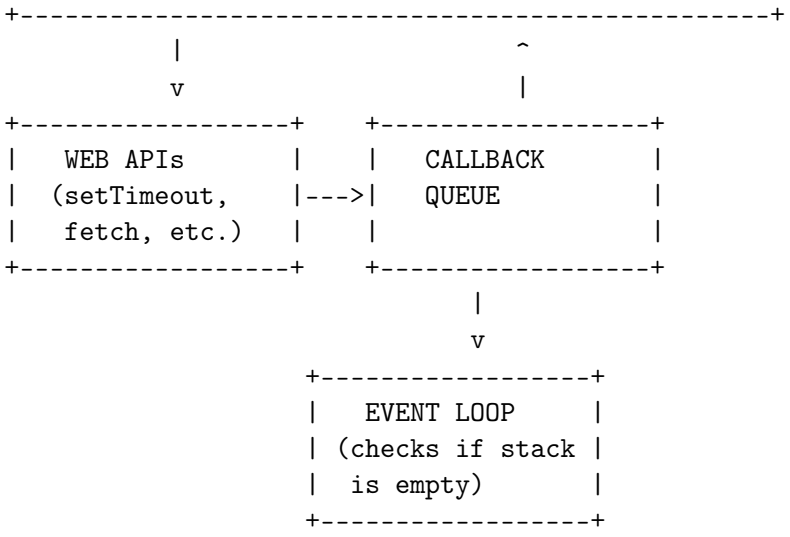
Visual Analogy

```
CONCURRENCY (Single Chef, Multiple Dishes)
+-----------------------------------------+
| [eggs] Start -> [toast] Start ->        |
| [eggs] Flip  -> [toast] Butter ->       |
| [eggs] Plate -> [toast] Plate           |
+-----------------------------------------+

PARALLELISM (Multiple Chefs, Multiple Dishes)
+-----------------------------------------+
| Chef 1: eggs  -> flip   -> plate        |
| Chef 2: toast -> butter -> plate        |
+-----------------------------------------+
```

JavaScript's Event Loop

```
+-------------------------------------------------+
|                     CALL STACK                  |
|  (synchronous, single-threaded execution)       |
```



JavaScript Context

```
// JavaScript is SINGLE-THREADED
// But can handle CONCURRENCY via event loop

// This is CONCURRENT (not parallel)
async function fetchData() {
  const [users, posts] = await Promise.all([
    fetch('/users'),
    fetch('/posts')
  ]);
}

// True PARALLELISM requires Web Workers
const worker = new Worker('heavy-task.js');
```

The Evolution of React Concurrency

Timeline: React 10-19

Version	Year	Key Concurrency Features
10-15	2014-2016	Stack reconciler (synchronous)
16	2017	Fiber architecture (foundation)
16.6	2018	Suspense for code splitting
17	2020	Gradual upgrades, event delegation
18	2022	Concurrent rendering, useTransition
19	2024	use() hook, Actions, improved Suspense
19.2	2025	Activity API, useEffectEvent

React 16: The Fiber Revolution (2017)

What Changed

- Complete rewrite of reconciliation algorithm
- Stack-based -> Fiber (linked list)
- Foundation for all future concurrency

Key Innovation: Interruptible Work

```
STACK RECONCILER (React 15):
  render() -> render() -> render() -> DONE
  [================================]
  Cannot stop once started

FIBER RECONCILER (React 16+):
  render() -> PAUSE -> render() -> PAUSE -> DONE
  [======]   yield   [======]    yield
  Can pause and resume
```

React 18: Concurrent Rendering (2022)

New Primitives

- `useTransition` - mark updates as non-urgent
- `useDeferredValue` - defer expensive values
- `useSyncExternalStore` - external store integration

- `useId` - stable IDs for SSR

Automatic Batching

```
// React 17: Only batched in event handlers
// React 18: Batched everywhere

setTimeout(() => {
  setCount(c => c + 1);  // Now batched
  setFlag(f => !f);      // Single re-render
}, 1000);
```

React 19: The use() Era (2024)

New Features

- `use()` hook for promises and context
- Actions (`useActionState`, `useFormStatus`)
- `useOptimistic` for instant feedback
- Document metadata (`<title>`, `<meta>`)
- Ref as prop (no more forwardRef)

React 19.2: Activity API (2025)

Activity Component

- Keep UI mounted but hidden
- Preserve state while unmounting effects
- Render with lower priority when hidden

```
<Activity mode={isActive ? "visible" : "hidden"}>
  <ExpensiveComponent />
</Activity>
```

Classic Concurrency Patterns

Throttle vs Debounce vs Defer

```
INPUT:     x]
TIME:      0----1----2----3----4----5----6----7

THROTTLE (max once per 2s):
OUTPUT:    x---------x---------x---------x
           ^         ^         ^         ^
           first     allowed   allowed   allowed

DEBOUNCE (wait 2s after last):
OUTPUT:    ----------------------------x
                                       ^
                                       fires after quiet period

DEFER (React's useDeferredValue):
OUTPUT:    x-x-x-x---x---------x-------x
           ^         ^         ^       ^
           immediate but yields to urgent work
```

Throttle Implementation

```
function throttle<T extends (...args: any[]) => any>(
  fn: T,
  limit: number
): (...args: Parameters<T>) => void {
  let lastCall = 0;

  return (...args: Parameters<T>) => {
    const now = Date.now();
    if (now - lastCall >= limit) {
      lastCall = now;
      fn(...args);
    }
  };
}

// Usage: max once per 100ms
const throttledScroll = throttle(handleScroll, 100);
```

```
window.addEventListener('scroll', throttledScroll);
```

Debounce Implementation

```
function debounce<T extends (...args: any[]) => any>(
  fn: T,
  delay: number
): (...args: Parameters<T>) => void {
  let timeoutId: ReturnType<typeof setTimeout>;

  return (...args: Parameters<T>) => {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => fn(...args), delay);
  };
}

// Usage: wait 300ms after typing stops
const debouncedSearch = debounce(search, 300);
input.addEventListener('input', debouncedSearch);
```

React's Built-in Alternative

```
// Instead of manual debounce, use useDeferredValue
function SearchResults({ query }: { query: string }) {
  const deferredQuery = useDeferredValue(query);

  // Advantages over debounce:
  // 1. No fixed delay - renders ASAP when idle
  // 2. Interruptible - yields to urgent updates
  // 3. Shows stale content (no blank states)

  const isStale = query !== deferredQuery;

  return (
    <div style={{ opacity: isStale ? 0.7 : 1 }}>
      <Results query={deferredQuery} />
    </div>
  );
}
```

CSP and core.async: A Different Model

Communicating Sequential Processes (CSP)

Origin

- Tony Hoare, 1978
- Foundation for Go's goroutines
- ClojureScript's core.async

Core Idea

```
+----------+                    +----------+
| Process  |                    | Process  |
|    A     |---> [Channel] --->|    B     |
|          |                    |          |
+----------+                    +----------+

- Processes are independent
- Communicate via channels
- Blocking put/take operations
- No shared mutable state
```

core.async in ClojureScript

Channels and Go Blocks

```
(ns demo.core
  (:require [cljs.core.async :refer [chan go <! >! timeout]]))

;; Create a channel
(def clicks (chan))

;; Producer: put clicks on channel
(defn on-click [event]
  (go (>! clicks event)))

;; Consumer: process clicks
(go
  (loop []
    (let [event (<! clicks)]
```

```
      (println "Clicked:" (.-target event))
      (recur))))
```

Debounce with core.async

```
(ns demo.debounce
  (:require [cljs.core.async :refer [chan go <! >! timeout alts!]]))

(defn debounce [in-ch ms]
  (let [out-ch (chan)]
    (go
      (loop [last-val nil]
        (let [[val ch] (alts! [in-ch (timeout ms)])]
          (condp = ch
            in-ch   (recur val)           ; New value, restart timer
            :else   (when last-val        ; Timeout, emit value
                      (>! out-ch last-val)
                      (recur nil))))))
    out-ch))

;; Usage
(def input-ch (chan))
(def debounced-ch (debounce input-ch 300))

(go
  (loop []
    (let [val (<! debounced-ch)]
      (search! val)
      (recur))))
```

Throttle with core.async

```
(ns demo.throttle
  (:require [cljs.core.async :refer [chan go <! >! timeout]]))

(defn throttle [in-ch ms]
  (let [out-ch (chan)]
    (go
      (loop []
        (let [val (<! in-ch)]
```

```
          (>! out-ch val)
          (<! (timeout ms))  ; Wait before allowing next
          (recur))))
    out-ch))

;; Usage
(def scroll-ch (chan))
(def throttled-ch (throttle scroll-ch 100))

(go
  (loop []
    (let [pos (<! throttled-ch)]
      (update-ui! pos)
      (recur))))
```

Comparing Approaches

Aspect	React Concurrent	core.async
Model	Priority lanes	CSP channels
Scheduling	Framework-managed	Developer-controlled
Cancellation	Automatic (re-render)	Manual
Mental model	Declarative	Imperative processes
Best for	UI responsiveness	Complex async flows

Why Concurrency Matters in React

The Problem: Blocking Renders

```
// Without concurrency - UI freezes
function App() {
  const [items, setItems] = useState([]);

  // BAD: Typing is blocked while list renders
  return (
    <>
      <SearchInput onChange={filterItems} />
      <HugeList items={items} /> {/* 10,000 items */}
    </>
  );
}
```

The Solution: Interruptible Rendering

```
// With React 19 concurrency - UI stays responsive
function App() {
  const [items, setItems] = useState([]);
  const deferredItems = useDeferredValue(items);

  // GOOD: Input responds instantly
  // GOOD: List renders when there's time
  return (
    <>
      <SearchInput onChange={filterItems} />
      <HugeList items={deferredItems} />
    </>
  );
}
```

React 19 Concurrency Primitives

Overview

Primitive	Purpose	Priority
`<Suspense>`	Async loading boundaries	Normal
`<Activity>`	Show/hide without unmount	Variable
`useTransition`	Mark updates as non-urgent	Low
`useDeferredValue`	Defer expensive computations	Low
`use()`	Unwrap promises in render	Normal

`<Suspense>`

What It Does

- Declarative loading states
- Catches "suspended" components
- Shows fallback while waiting

Example

```
import { Suspense } from 'react';
```

```
function App() {
  return (
    <Suspense fallback={<Skeleton />}>
      <UserProfile />  {/* May suspend */}
    </Suspense>
  );
}

// Component that suspends
function UserProfile() {
  const user = use(fetchUser()); // Suspends until resolved
  return <div>{user.name}</div>;
}
```

`<Activity>`

What It Does

- Formerly called `<Offscreen>`
- Preserves state when hidden
- Pre-renders content at low priority
- Like CSS `visibility: hidden` but smarter

Modes

Mode	Rendered	Visible	Priority
`visible`	Yes	Yes	Normal
`hidden`	Yes	No	Low

Example: Tabs Without State Loss

```
import { Activity } from 'react';

function TabContainer({ activeTab }) {
  return (
    <>
      <Activity mode={activeTab === 'home' ? 'visible' : 'hidden'}>
        <HomePage />  {/* State preserved when hidden */}
      </Activity>
```

```
      <Activity mode={activeTab === 'profile' ? 'visible' : 'hidden'}>
        <ProfilePage />  {/* Pre-rendered, ready instantly */}
      </Activity>
    </>
  );
}
```

Example: Pre-rendering

```
function App() {
  const [showModal, setShowModal] = useState(false);

  return (
    <>
      <button onClick={() => setShowModal(true)}>
        Open Settings
      </button>

      {/* Pre-render modal at low priority */}
      <Activity mode={showModal ? 'visible' : 'hidden'}>
        <SettingsModal onClose={() => setShowModal(false)} />
      </Activity>
    </>
  );
}
```

`useTransition`

What It Does

- Marks state updates as non-urgent
- Keeps UI responsive during updates
- Provides `isPending` for loading states

Example

```
import { useTransition } from 'react';

function SearchResults() {
```

```
  const [query, setQuery] = useState('');
  const [results, setResults] = useState([]);
  const [isPending, startTransition] = useTransition();

  function handleSearch(e) {
    // Urgent: update input immediately
    setQuery(e.target.value);

    // Non-urgent: filter can wait
    startTransition(() => {
      setResults(filterLargeDataset(e.target.value));
    });
  }

  return (
    <>
      <input value={query} onChange={handleSearch} />
      {isPending && <Spinner />}
      <ResultsList results={results} />
    </>
  );
}
```

`useDeferredValue`

What It Does

- Defers updating a value
- Simpler than `useTransition`
- Good for derived/computed values

Example

```
import { useDeferredValue, useMemo } from 'react';

function FilteredList({ items, filter }) {
  // Deferred version of filter
  const deferredFilter = useDeferredValue(filter);

  // Expensive computation uses deferred value
```

```
  const filteredItems = useMemo(
    () => items.filter(item =>
      item.name.includes(deferredFilter)
    ),
    [items, deferredFilter]
  );

  // Show stale indicator
  const isStale = filter !== deferredFilter;

  return (
    <div style={{ opacity: isStale ? 0.7 : 1 }}>
      {filteredItems.map(item => (
        <Item key={item.id} {...item} />
      ))}
    </div>
  );
}
```

`use()` Hook

What It Does

- New in React 19
- Unwraps promises and context
- Can be called conditionally
- Suspends component until resolved

Example with Promises

```
import { use, Suspense } from 'react';

// Can be created outside component
const dataPromise = fetch('/api/data').then(r => r.json());

function DataDisplay() {
  // Suspends until promise resolves
  const data = use(dataPromise);
```

```
  return <div>{data.title}</div>;
}

function App() {
  return (
    <Suspense fallback={<Loading />}>
      <DataDisplay />
    </Suspense>
  );
}
```

Example with Context

```
import { use } from 'react';

function ConditionalTheme({ showTheme }) {
  // Can call conditionally (unlike useContext)
  if (showTheme) {
    const theme = use(ThemeContext);
    return <div style={{ color: theme.primary }}>Themed!</div>;
  }
  return <div>No theme</div>;
}
```

Combining Primitives

Pattern: Responsive Tabs

```
function App() {
  const [activeTab, setActiveTab] = useState('home');
  const [isPending, startTransition] = useTransition();

  function switchTab(tab) {
    startTransition(() => {
      setActiveTab(tab);
    });
  }

  return (
    <>
```

```
      <TabBar
        active={activeTab}
        onChange={switchTab}
        loading={isPending}
      />

      <Activity mode={activeTab === 'home' ? 'visible' : 'hidden'}>
        <Suspense fallback={<HomeSkeleton />}>
          <HomePage />
        </Suspense>
      </Activity>

      <Activity mode={activeTab === 'search' ? 'visible' : 'hidden'}>
        <Suspense fallback={<SearchSkeleton />}>
          <SearchPage />
        </Suspense>
      </Activity>
    </>
  );
}
```

Pattern: Optimistic Updates

```
import { useOptimistic, useTransition } from 'react';

function LikeButton({ postId, initialLikes }) {
  const [likes, setLikes] = useState(initialLikes);
  const [optimisticLikes, addOptimistic] = useOptimistic(
    likes,
    (current, increment) => current + increment
  );
  const [isPending, startTransition] = useTransition();

  async function handleLike() {
    addOptimistic(1); // Instant UI update

    startTransition(async () => {
      const newCount = await likePost(postId);
      setLikes(newCount); // Sync with server
    });
```

```
  }

  return (
    <button onClick={handleLike} disabled={isPending}>
      Like {optimisticLikes}
    </button>
  );
}
```

How React Scheduler Works

Time Slicing

```
Traditional Rendering:
+----------------------------------------+
| Render Component Tree (blocks 200ms)   |
+----------------------------------------+
BAD: User input ignored for 200ms

Concurrent Rendering:
+------+ +------+ +------+ +------+
|Render| |Yield | |Render| |Yield | ...
+------+ +------+ +------+ +------+
           ^ Check for user input
GOOD: UI stays responsive
```

Priority Lanes

```
React schedules work in priority lanes:

+------------------+----------------------+
| Sync Lane        | Click handlers       | <- Immediate
+------------------+----------------------+
| Input Lane       | Typing, key presses  | <- Very High
+------------------+----------------------+
| Default Lane     | Normal state updates | <- Normal
+------------------+----------------------+
| Transition Lane  | useTransition        | <- Low
+------------------+----------------------+
| Idle Lane        | Activity hidden      | <- Very Low
```

```
+-----------------+---------------------+
```

Best Practices

When to Use What

Scenario	Solution
Loading async data	`<Suspense>` + `use()`
Tabs/routes state preserved	`<Activity>`
Expensive list filtering	`useDeferredValue`
Non-urgent state update	`useTransition`
Pre-render likely-needed UI	`<Activity mode="hidden">`
Instant feedback	`useOptimistic`

Performance Tips

1. **Wrap expensive subtrees** in `<Activity>` or `Suspense`
2. **Use transitions** for updates that don't need instant feedback
3. **Defer values** that drive expensive computations
4. **Pre-render** content users are likely to see
5. **Don't over-optimize** - profile first!

Common Mistakes

```
// BAD: Transition for urgent updates
startTransition(() => {
  setInputValue(e.target.value); // User expects instant feedback
});

// GOOD: Transition for non-urgent updates
setInputValue(e.target.value); // Instant
startTransition(() => {
  setSearchResults(search(e.target.value)); // Can wait
});

// BAD: Suspense without fallback
<Suspense>
  <AsyncComponent />
```

```
</Suspense>

// GOOD: Always provide fallback
<Suspense fallback={<Skeleton />}>
  <AsyncComponent />
</Suspense>
```

Summary

Key Takeaways

1. **Concurrency** = managing multiple tasks (structure)
2. **Parallelism** = executing multiple tasks (execution)
3. React uses **concurrency** to keep UI responsive
4. **Interruptible rendering** lets urgent updates cut in line
5. Use the right primitive for the job

The Mental Model

```
User clicks button while list renders:

OLD REACT:
  [========= Render List =========] -> [Handle Click]
  300ms delay (bad)

REACT 19:
  [=== Render ===] PAUSE [Click!] -> [=== Resume ===]
  Instant response (good)
```

Resources

- React 19 Blog Post
- React 18 Release Notes
- Suspense Documentation
- useTransition Documentation

- React Working Group Discussions
- Clojure core.async Channels
- Mastering core.async

Reagent and re-frame (ClojureScript)

Reagent: ClojureScript + React

What Is It?

- Minimalistic React wrapper for ClojureScript
- Hiccup syntax instead of JSX
- Reactive atoms (RAtoms) for state
- Predates React Hooks by years

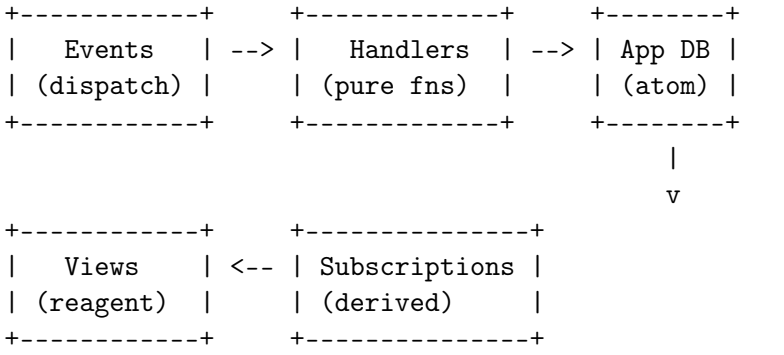
Example Component

```
(def counter (r/atom 0))

(defn counter-view []
  [:div
   [:p "Count: " @counter]
   [:button {:on-click #(swap! counter inc)} "+"]])
```

re-frame: Unidirectional Data Flow

Architecture



Key Concepts

```
;; Event handler (pure function)
(rf/reg-event-db :increment
  (fn [db [_ amount]]
    (update db :count + amount)))

;; Subscription (derived data)
(rf/reg-sub :count
  (fn [db _] (:count db)))

;; View (reactive)
(defn counter []
  (let [count @(rf/subscribe [:count])]
    [:button {:on-click #(rf/dispatch [:increment 1])}
     count]))
```

Current Status (2025)

Compatibility Matrix

React Version	Reagent Support	Notes
React 17	Full	Stable, production-ready
React 18	Partial	Legacy render API works
React 19	Incompatible	Class component conflicts

The Challenge

- Reagent built on class components
- React 19 deprecates class patterns
- Concurrent features inaccessible

Alternatives

- **UIx2**: Hooks-first wrapper
- **Helix**: Modern ClojureScript React
- **HSX/RFX**: Custom Hiccup + hooks

React 19 vs re-frame Patterns

Concept	React 19	re-frame
State	useState, useReducer	Single atom (app-db)
Derived data	useMemo	Subscriptions
Side effects	useEffect, Actions	reg-event-fx, reg-fx
Priority	useTransition	Not built-in
Async coordination	Suspense, use()	dispatch-n, effects

Practice Projects

10 Concurrency-Focused Projects

#	Project	Key Concurrency Pattern
1	Typeahead Search	useDeferredValue / debounce
2	Modal Dialog System	Activity pre-rendering
3	Infinite Scroll	Throttle + virtualization
4	Form Validation	Async validation + debounce
5	Image Carousel	Preloading + transitions
6	Notifications	Priority queuing
7	Data Table	Deferred sort/filter
8	Multi-step Wizard	Step pre-rendering
9	Collaborative Editor	Optimistic updates + merging
10	Live Dashboard	Staggered refresh + isolation

See `docs/projects.org` for Full Specs

Each project includes:

- Functional requirements
- Concurrency focus areas
- React 19 implementation hints
- Reagent/re-frame implementation hints
- Bonus challenges

Questions?

Thank you!
React Paris Meetup #011

Speakers: Ariel Shulman - Conquering React Concurrency Theo Senoussaoui - Evolution des librairies UI
Materials: github.com/jwalsh/react-paris-meetup-011