# OOM Must Fail-fast

## Stage 1 update


**Mark S. Miller** — Agoric

**Peter Hoddie** — moddable

**Zbyszek Tenerowicz** — METAMASK

**Christopher Hiller** — METAMASK



**107th Plenary**

**April 2025**

TC 39

# Don’t Remember Panicking

## Stage 1 update


**Mark S. Miller** — Agoric

**Peter Hoddie** — moddable

**Zbyszek Tenerowicz** — METAMASK

**Christopher Hiller** — METAMASK



**107th Plenary**

**April 2025**

TC 39

# Example: Prepare-Commit Pattern

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #units; #unitValue;
  constructor(units, unitValue) {
    this.#units = Nat(units); this.#unitValue = Nat(unitValue);
  }

  toString() { return `${this.#units} * ${this.#unitValue}`; }

  deposit(myDelta, src) {
    Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
    // COMMIT POINT
    this.#units += myDelta;
    src.#units -= myDelta * this.#unitValue / src.#unitValue;
  }
}
```

# Example: Prepare-Commit Pattern

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #units; #unitValue;
  constructor(units, unitValue) {
    this.#units = Nat(units); this.#unitValue = Nat(unitValue);
  }

  toString() { return `${this.#units} * ${this.#unitValue}`; }

  deposit(myDelta, src) {
    Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
    // COMMIT POINT
    this.#units += myDelta;
    src.#units -= myDelta * this.#unitValue / src.#unitValue;
  }
}
```

Transactional Totality
All effects or none.

# Example: Prepare-Commit Pattern

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #units; #unitValue;
  constructor(units, unitValue) {
    this.#units = Nat(units); this.#unitValue = Nat(unitValue);
  }

  toString() { return `${this.#units} * ${this.#unitValue}`; }

  deposit(myDelta, src) {
    Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
    // COMMIT POINT
    this.#units += myDelta;
    src.#units -= myDelta * this.#unitValue / src.#unitValue;
  }
}
```

**Prepare Phase**
All possible throws or early returns.
No effects.

# Example: Prepare-Commit Pattern

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #units; #unitValue;
  constructor(units, unitValue) {
    this.#units = Nat(units); this.
  }

  toString() { return `${this.#units} * ${this.#unitValue}`; }

  deposit(myDelta, src) {
    Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
    // COMMIT POINT
    this.#units += myDelta;
    src.#units -= myDelta * this.#unitValue / src.#unitValue;
  }
}
```

Prepare Phase

All possible throws or early returns.
No effects.

- Are `this` and `src` instances of `Purse`?
- Is `myDelta` non-negative?
- Would `src` be overdrawn?

# Example: Prepare-Commit Pattern

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #units; #unitValue;
  constructor(units, unitValue) {
    this.#units = Nat(units); this.#unitValue = Nat(unitValue);
  }

  toString() { return `${this.#units} * ${this.#unitValue}`; }

  deposit(myDelta, src) {
    Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
    // COMMIT POINT
    this.#units += myDelta;
    src.#units -= myDelta * this.#unitValue / src.#unitValue;
  }
}
```

**Prepare Phase**
All possible throws or early returns.
No effects.

**Fragile Phase**
No throws.
All effects.

# Recap: OOM or OOS

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #units; #unitValue;
  constructor(units, unitValue) {
    this.#units = Nat(units); this.#unitValue = Nat(unitValue);
  }

  toString() { return `${this.#units} * ${this.#unitValue}`; }

  deposit(myDelta, src) {
    Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
    // COMMIT POINT
    this.#units += myDelta;
    src.#units -= myDelta * this.#unitValue / src.#unitValue; // Zalgo beckons
  }
}
```

OOM or OOS almost anywhere.
Always unexpected.
Corrupted state not repairable.

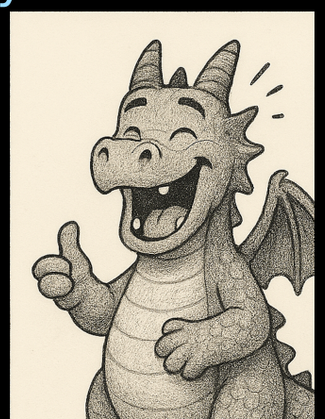
# Recap: How to repair in general?

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #units; #unitValue;
  constructor(units, unitValue) {
    this.#units = Nat(units); this.#unitValue = Nat(unitValue);
  }

  toString() { return `${this.#units} * ${this.#unitValue}`; }

  deposit(myDelta, src) {
    Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
    try { // COMMIT POINT
      this.#units += myDelta;
      src.#units -= myDelta * this.#unitValue / src.#unitValue;
    } catch (err) {
      // Zalgo laughs
    }
  }
}
```



# Recap: Need Internal Panic

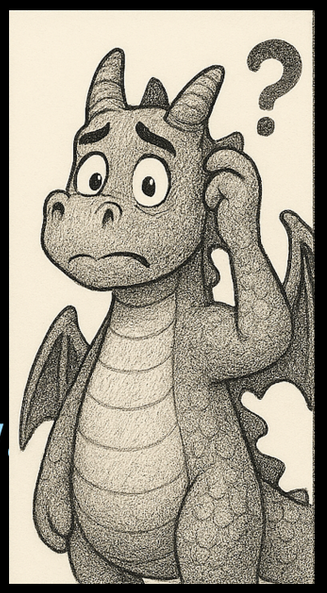
```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #units; #unitValue;
  constructor(units, unitValue) {
    this.#units = Nat(units); this.#unitValue = Nat(unitValue);
  }

  toString() { return `${this.#units} * ${this.#unitValue}`; }

  deposit(myDelta, src) {
    Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitV
    // COMMIT POINT
    this.#units += myDelta;
    src.#units -= myDelta * this.#unitValue / src.#unitValue; // Zalgo forgets
  }
}
```

*Possible* Host OOM Policy:
Exit Agent(?) Immediately

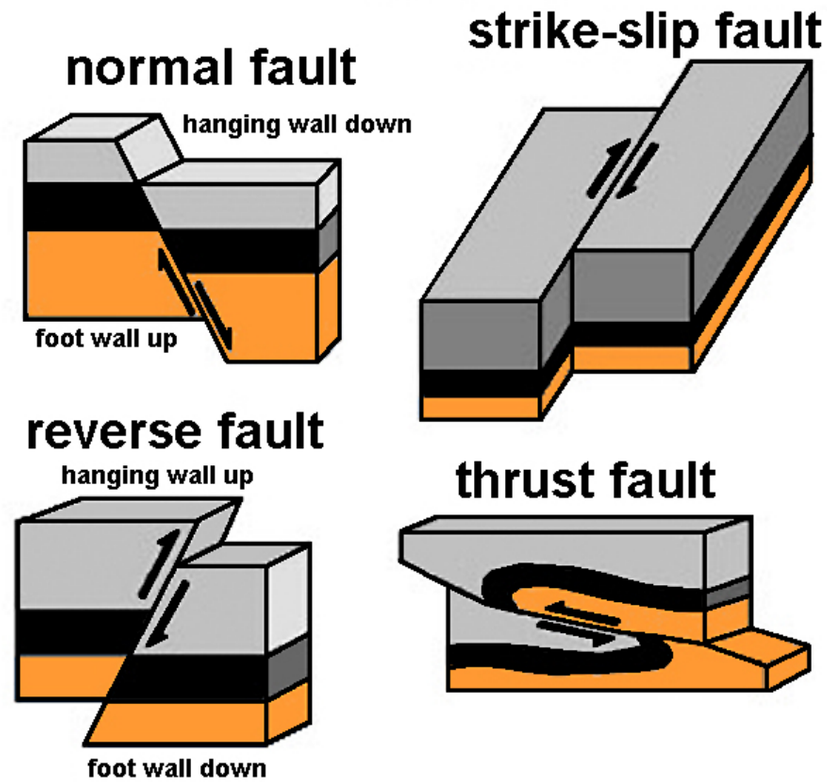


# “Trusted” Host Hook

```
HostFaultHandler(faultType, arg=undefined)
```

# Fault Taxonomy

```
HostFaultHandler(faultType, arg=undefined)
```



# Fault Taxonomy

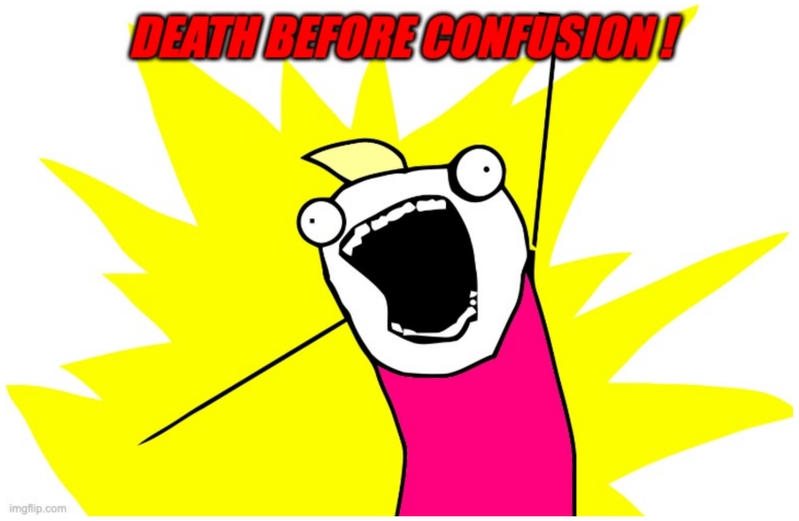
```
HostFaultHandler(faultType, arg=undefined)
```

Unrepairable corruption
!(Availability & Integrity)

# Fault Taxonomy

`HostFaultHandler(faultType, arg=undefined)`

Unrepairable corruption
!(Availability & Integrity)

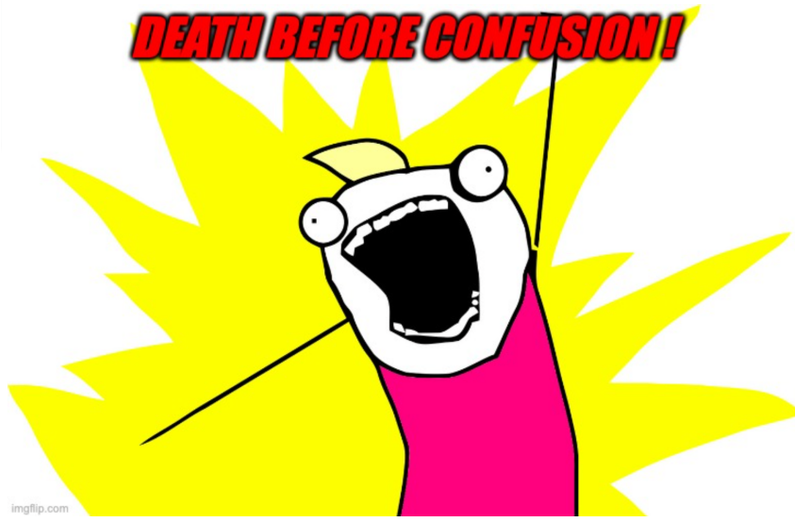




Fail-stop
sacrifice Availability for Integrity

# Fault Taxonomy

```
HostFaultHandler(faultType, arg=undefined)
```

Unrepairable corruption
!(Availability & Integrity)


Fail-stop
sacrifice Availability for Integrity

Best-efforts
sacrifice Integrity for Availability

# Fault Taxonomy

```
HostFaultHandler(faultType, arg=undefined)
```

**Host internal invariant violated.**
**Unrepairable host corrupted state.**
**Must fail-stop.**
**Blue tabs of death. Reboot device. …**

| | XS | Browser? |
|---|---|---|
| | OOM | |
| | OOS | |
| | InternalAssert | InternalAssert |
| | NoMoreKeys (xs) | |

# Fault Taxonomy

```
HostFaultHandler(faultType, arg=undefined)
```

| | XS | Browser? |
|---|---|---|
| Host internal invariant violated.<br>Unrepairable host corrupted state.<br>Must fail-stop.<br>Blue tabs of death. Reboot device. … | OOM<br>OOS<br>InternalAssert<br>NoMoreKeys (xs) | InternalAssert |
| Host cannot proceed within JS spec.<br>Unrepairable JS corrupted state.<br>Best-efforts vs fail-stop.<br>JS opt-in to fail-stop. | DeadStrip (xs) | OOM<br>OOS |
| | | |
| | | |

# Fault Taxonomy

`HostFaultHandler(faultType, arg=undefined)`

Host internal invariant violated.
Unrepairable host corrupted state.
Must fail-stop.
Blue tabs of death. Reboot device. …

Host cannot proceed within JS spec.
Unrepairable JS corrupted state.
Best-efforts vs fail-stop.
JS opt-in to fail-stop.

**Host fine.**
**JS code in trouble?**
**Best-efforts vs fail-stop.**

| XS | Browser? |
|---|---|
| OOM | |
| OOS | |
| InternalAssert | InternalAssert |
| NoMoreKeys (xs) | |
| DeadStrip (xs) | OOM |
| | OOS |
| Debugger (xs) | |
| OO Time | OO Time |
| UnhandledException | UnhandledException |
| UnhandledRejection | UnhandledRejection |

whatwg / html

Type / to search

# Shared memory and deactivated agents #2581

Open

**annevk** opened on Apr 25, 2017

This is a follow-up to #2521. JavaScript has the following requirement on agents:

> An embedding may terminate an agent without any of the agent's cluster's other agents' prior knowledge or cooperation. If an agent is terminated not by programmatic action of its own or of another agent in the cluster but by forces external to the cluster, then the embedding must choose one of two strategies: Either terminate all the agents in the cluster, or provide reliable APIs that allow the agents in the cluster to coordinate so that at least one remaining member of the cluster will be able to detect the termination, with the termination data containing enough information to identify the agent that was terminated.

Boris came up with a scenario that doesn't quite meet this and I found one derived from that:

# Agent cluster and crash recovery: specifications? #4901

Open

**gterzian** opened on Sep 11, 2019 · edited by gterzian  Edits

Are there existing efforts, and if not, should there be, to standardize the "recover from a crashed tab" workflow? Also, this could be broadened to "how to recover from a crashed agent-cluster?".

Assignee

No one a

Labels

# Discuss out-of-memory etc. agent cluster killing, especially for shared workers? #11205

Open

**domenic** opened 2 days ago

**What is the issue with the HTML Standard?**

Currently, the HTML Standard does not really discuss the fact that operating systems or browser implementations can kill processes, e.g. due to memory pressure. (The closest I can find is the killing scripts section.)

Assignees

No one assigned

Labels

topic: agent  topic: w

# (Scroll to test code)

```
const Nat = n => {
  if (n >= 0) { return n; }
  throw RangeError(`not Nat ${n}`);
};

class Purse {
  #units; #unitValue;
  constructor(units, unitValue) {
    this.#units = Nat(units); this.#unitValue = Nat(unitValue);
  }

  toString() { return `${this.#units} * ${this.#unitValue}`; }

  deposit(myDelta, src) {
    Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
    // COMMIT POINT
    this.#units += myDelta;
    src.#units -= myDelta * this.#unitValue / src.#unitValue;
  }
}
```

# No bug seen during dev,review,tests

```
  deposit(myDelta, src) {
    Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
    // COMMIT POINT
    this.#units += myDelta;
    src.#units -= myDelta * this.#unitValue / src.#unitValue; // Zalgo smirks
  }
}
```

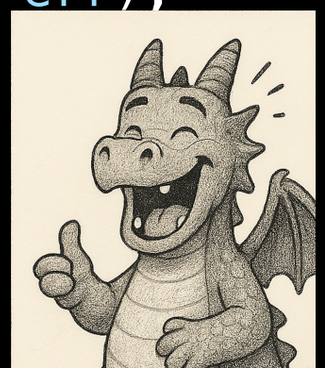
```
const mine = new Purse(10, 1000);
const src = new Purse(20, 500);
try {
  mine.deposit(3, src);
  console.log(`${mine}, ${src}`); // 13 * 1000, 14 * 500
} catch (err) {
  console.log('caught', err);     // Not reached. No bug seen.
}
```

# Triggered by Rare Data

```
  deposit(myDelta, src) {
    Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
    // COMMIT POINT
    this.#units += myDelta;
    src.#units -= myDelta * this.#unitValue / src.#unitValue;
  }
}
```

VM ok.
“Just” an unknown user bug

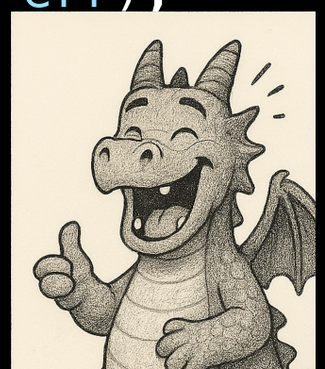
```
const mine = new Purse(10n, 1000n);
const src = new Purse(20n, 0n);
try {
  mine.deposit(0n, src);
  console.log(`${mine}, ${src}`); // not reached
} catch (err) {
  console.log('caught', err);     // caught RangeError: Division by zero
  // Zalgo laughs again
}
```



# At least we know why Zalgo laughs

```
deposit(myDelta, src) {
  Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
  try { // COMMIT POINT
    this.#units += myDelta;
    src.#units -= myDelta * this.#unitValue / src.#unitValue;
  } catch (err) {
    console.log('Zalgo', err); // Zalgo laughs
    throw err;
  }
}

const mine = new Purse(10n, 1000n);
const src = new Purse(20n, 0n);
try {
  mine.deposit(0n, src);          // Zalgo RangeError: Division by zero
  console.log(`${mine}, ${src}`); // not reached
} catch (err) {
  console.log('caught', err);     // caught RangeError: Division by zero
  // Zalgo laughs again
}
```



# Expensive defense possible today

```
deposit(myDelta, src) {
  Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
  try { // COMMIT POINT
    this.#units += myDelta;
    src.#units -= myDelta * this.#unitValue / src.#unitValue;
  } catch (err) {
    for (;;) {}
  }
}
```

```
const mine = new Purse(10n, 1000n);
const src = new Purse(20n, 0n);
try {
  mine.deposit(0n, src);          // Zalgo sleeps
  console.log(`${mine}, ${src}`); // not reached
} catch (err) {
  console.log('caught', err);     // not reached
}
```

# Somebody stop me!

```
  deposit(myDelta, src) {
    Nat(src.#units * src.#unitValue - Nat(myDelta) * this.#unitValue);
    try { // COMMIT POINT
      this.#units += myDelta;
      src.#units -= myDelta * this.#unitValue / src.#unitValue;
    } catch (err) {
      Reflect.panic(err);
    }
  }
}
```

```
const mine = new Purse(10n, 1000n);
const src = new Purse(20n, 0n);
try {
  mine.deposit(0n, src);          // Zalgo forgets
  console.log(`${mine}, ${src}`); // not reached
} catch (err) {
  console.log('caught', err);     // not reached
}
```

# Full Fault Taxonomy

`HostFaultHandler(faultType, arg=undefined)`

Host internal invariant violated.
Unrepairable host corrupted state.
Must fail-stop.
Blue tabs of death. Reboot device. …

Host cannot proceed within JS spec.
Unrepairable JS corrupted state.
Best-efforts vs fail-stop.
JS opt-in to fail-stop.

Host fine.
JS code in trouble?
Best-efforts vs fail-stop.

| XS | Browser? |
|---|---|
| OOM | |
| OOS | |
| InternalAssert | InternalAssert |
| NoMoreKeys (xs) | |
| DeadStrip (xs) | OOM |
| | OOS |
| Debugger (xs) | |
| OO Time | OO Time |
| UnhandledException | UnhandledException |
| UnhandledRejection | UnhandledRejection |

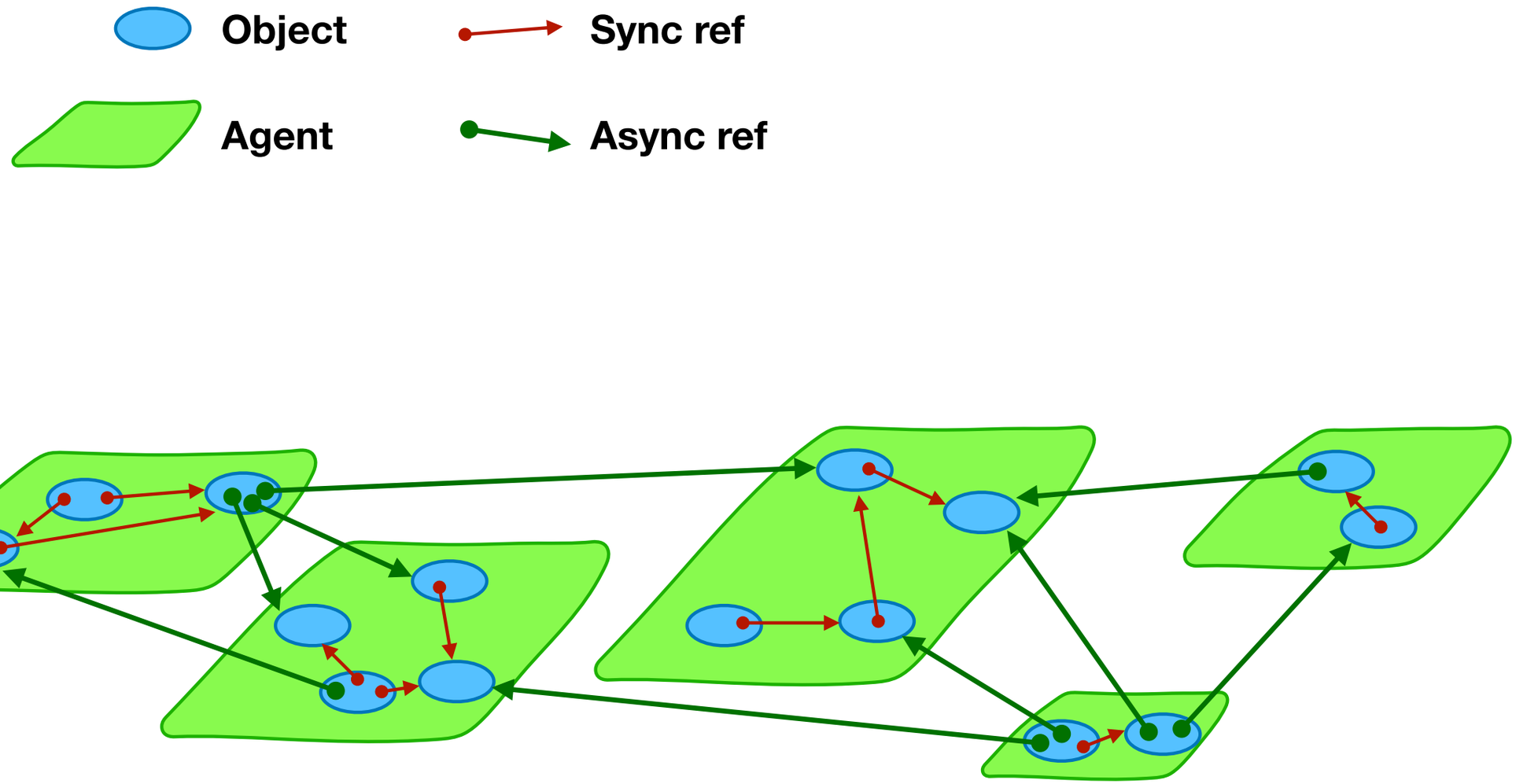
# Full Fault Taxonomy

`HostFaultHandler(faultType, arg=undefined)`

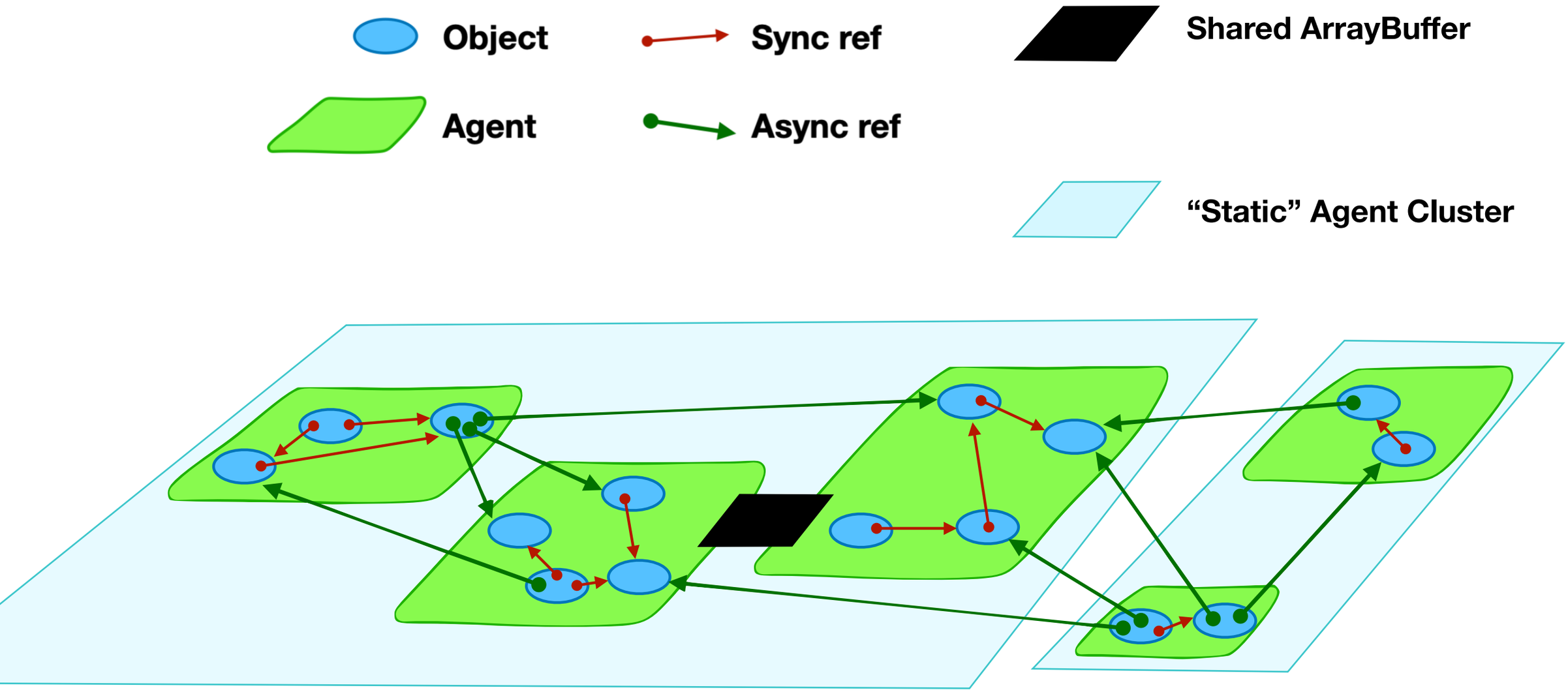
| | XS | Browser? |
|---|---|---|
| Host internal invariant violated.<br>Unrepairable host corrupted state.<br>Must fail-stop.<br>Blue tabs of death. Reboot device. … | `OOM`<br>`OOS`<br>`InternalAssert`<br>`NoMoreKeys (xs)` | `InternalAssert` |
| Host cannot proceed within JS spec.<br>Unrepairable JS corrupted state.<br>Best-efforts vs fail-stop.<br>JS opt-in to fail-stop. | `DeadStrip (xs)` | `OOM`<br>`OOS` |
| Host fine.<br>JS code in trouble?<br>Best-efforts vs fail-stop. | `Debugger (xs)`<br>`OO Time`<br>`UnhandledException`<br>`UnhandledRejection` | `OO Time`<br>`UnhandledException`<br>`UnhandledRejection` |
| **JS code notices own corruption.<br>Explicit JS `panic()` must fail-stop.** | `UserPanic` | |

```
Reflect.panic(arg=undefined);
```

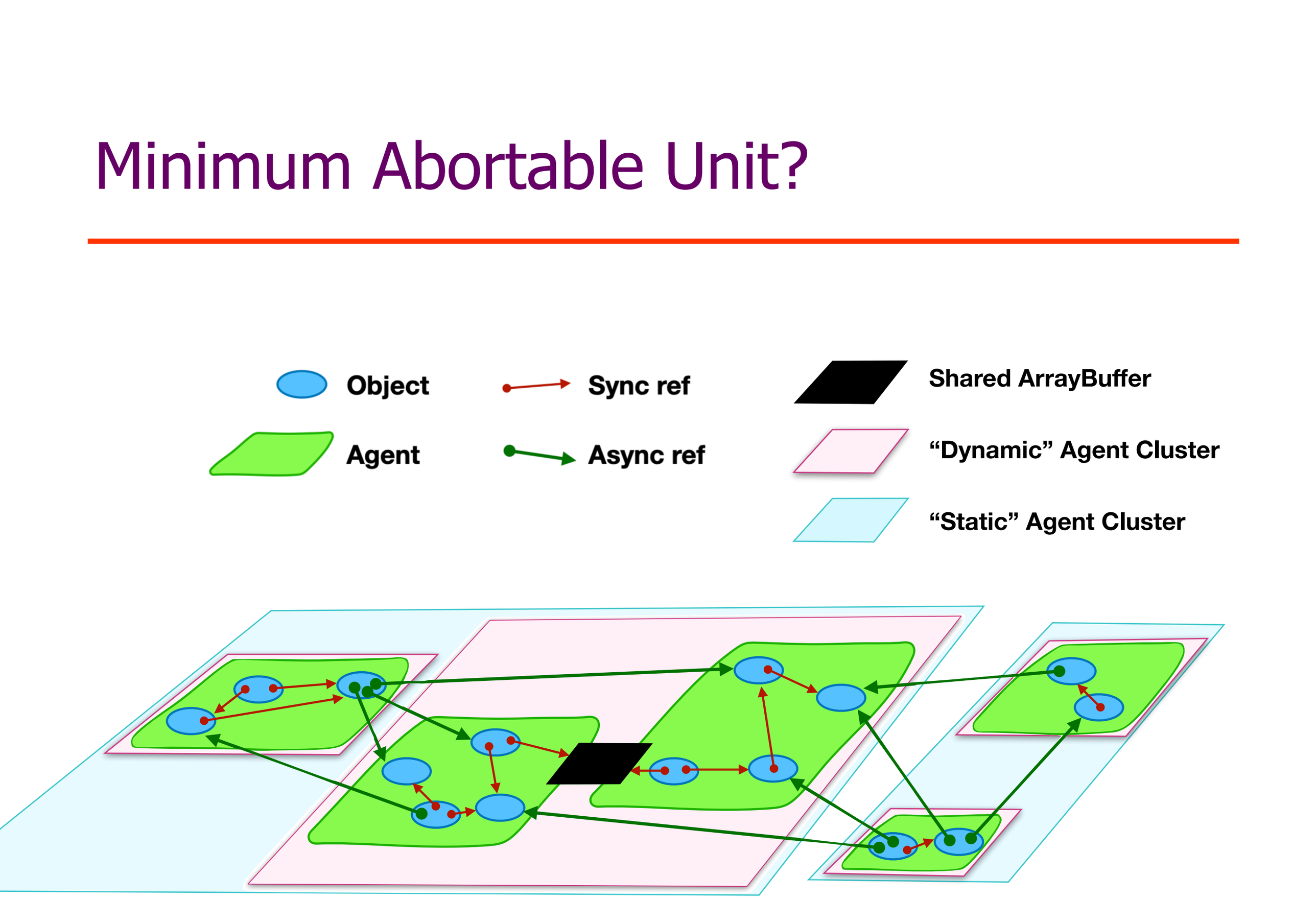
# Minimum Abortable Unit?



# Minimum Abortable Unit?



# Minimum Abortable Unit?



# Ways Host Hook Might Resume

```
return …;

throw …;
```

# Ways Host Hook Might Resume

```
return …;

throw …;

keeper(…);
```

delegate to user fault handler
(KeyKOS keeper, Erlang supervisor)
generalize “post-mortem” philosophy

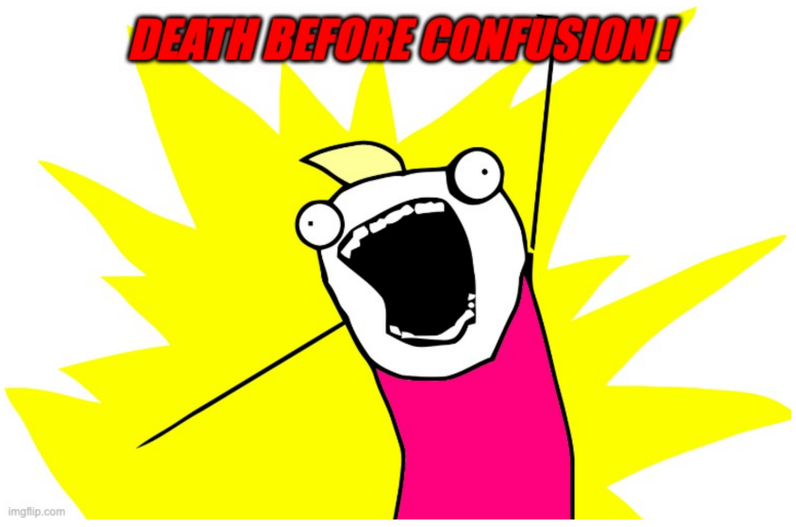

```
function keeper(…) {}
const agent = new Agent(…, {keeper});
```

# Ways Host Hook Might Not Resume

Coredump or other diagnostic snapshot

Refresh page (browser)

Blue Tabs of Death (browser)



# Ways Host Hook Might Not Resume

Coredump or other diagnostic snapshot

Refresh page (browser)

Blue Tabs of Death (browser)

Reboot device (XS, ***Pacemaker story***)

Abort transaction (Agoric: restore prev consistent state)

# Fault Tolerant Systems of…

Byzantine components: hard

- Fault masking: BFT, Blockchain.
- Useless against replicated bugs.

Fail-stop components: comparatively easy

- Fault masking: Simple redundancy. Tandem non-stop.

- Fault containment. Recovery via Keepers

  Lessons from Erlang, KeyKOS, EROS.

# Fault Tolerant Systems of…

## Byzantine components: hard

- Fault masking: BFT, Blockchain.
- Useless against replicated bugs.

## Fail-stop components: comparatively easy

- Fault masking: Simple redundancy. Tandem non-stop.
- Fault containment. Recovery via Keepers

  Lessons from Erlang, KeyKOS, EROS.

**Agoric code running on Agoric blockchain**

# Questions?