A Characterization of State Spill in Modern Operating Systems

Kevin Boos, Emilio Del Vecchio, and Lin Zhong

Advanced OS goals are challenging

Goal in OS literature	Impediments to that goal
Process migration	Residual dependencies on original system
Fault isolation/tolerance, software virtualization	Sprawl of states introduces fate sharing, complicates isolation & multiplexing logic
Live update and hot-swapping	Cannot modify individual entity in isolation; state transfer functions are non-trivial
Maintainability	Coupling remains despite modularization
Security	Loss of control over propagated data

State spill is the underlying cause

State spill is the act of a software entity's state **undergoing a lasting change** as a result of handling a transaction from another entity.

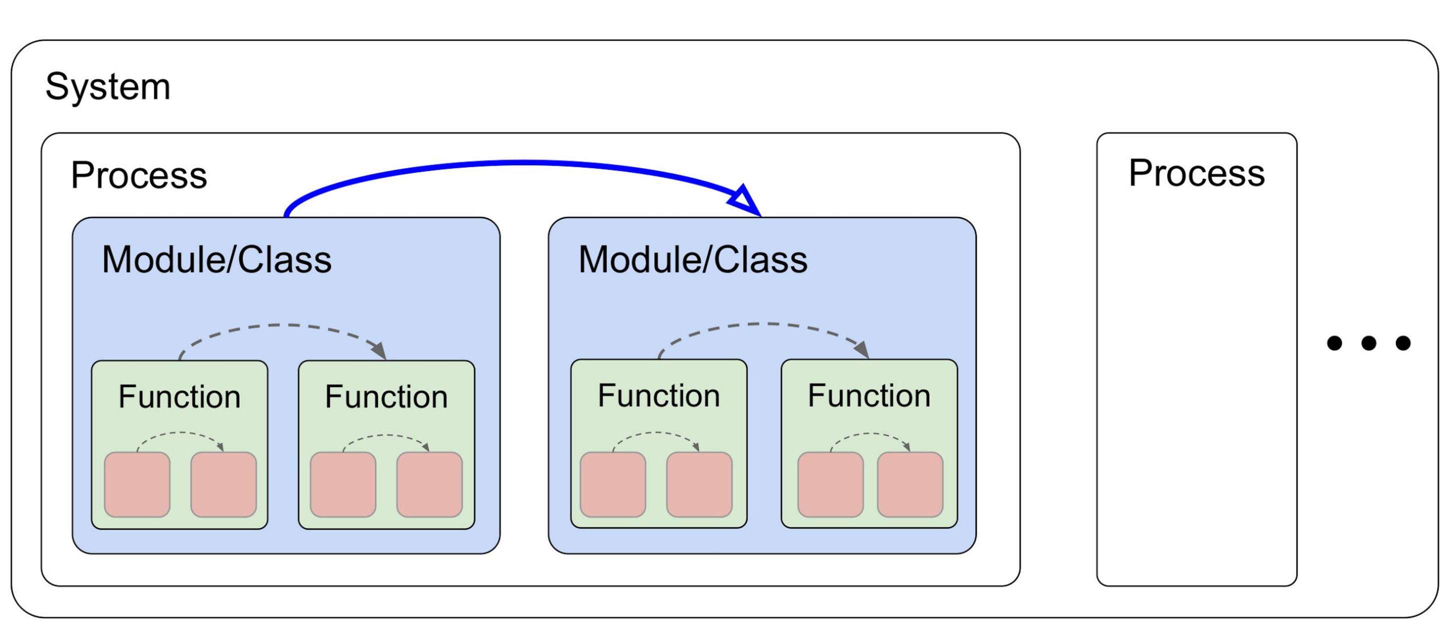
```
public class SystemService {
  static int sCount;
  byte mConfig;
  List<Callback> mCallbacks;
  int unrelated;

  public void addCallback(int id,
      byte cf, Callback cb) {
    int b = id;
    Log.print("id=" + b);
    this.mConfig = cf;
    this.mCallbacks.add(cb);
    sCount++;
  }
}
```

This method is a transaction handler invoked by application processes.

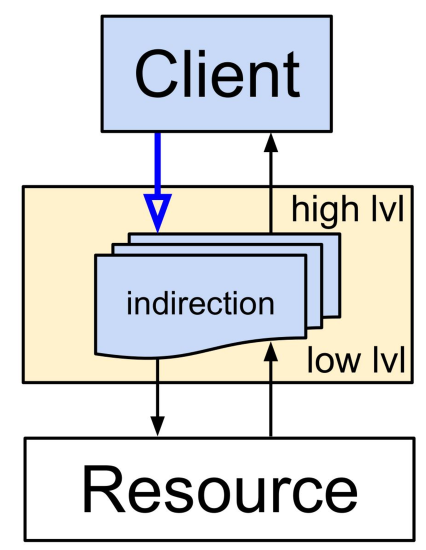
Entity granularity dictates state spill

State spill is relative to the chosen entity granularity. Low-level entity interactions (shaded) are unimportant.



Classification of state spill

Based on four common OS entity design patterns:

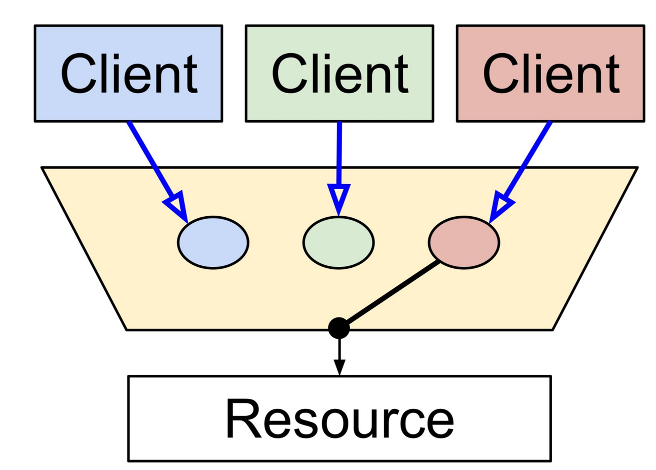


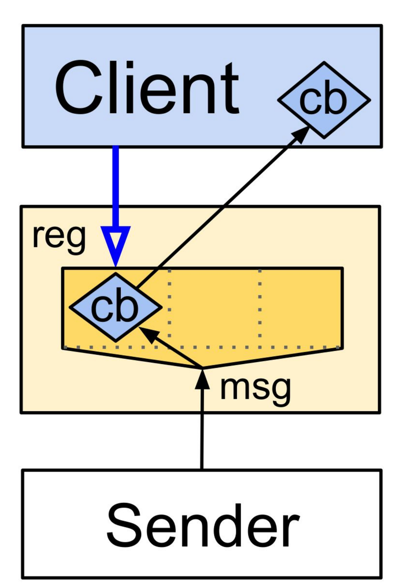
Indirection Layers convert between high-level and low-level representations of data and commands.

- Virtual File System abstraction
- Process abstraction
- Microkernel userspace servers
- Device drivers

Multiplexers temporally or spatially share an underlying resource among multiple clients.

- Schedulers / process mgmt
- Window managers
- High-level drivers



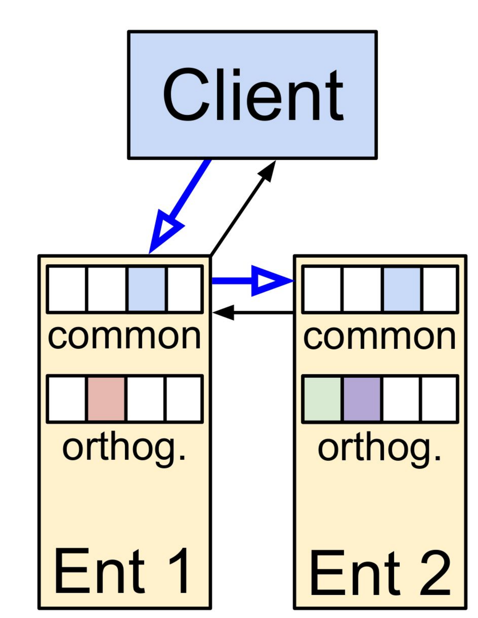


Dispatchers register client callbacks to properly deliver events or messages.

- Device event callbacks
- Synchronization primitives
- Upcalls
- IPC layers

Inter-Entity Collaboration requires synchronization of non-orthogonal states to ensure correctness.

- Microkernel userspace servers
- Android services

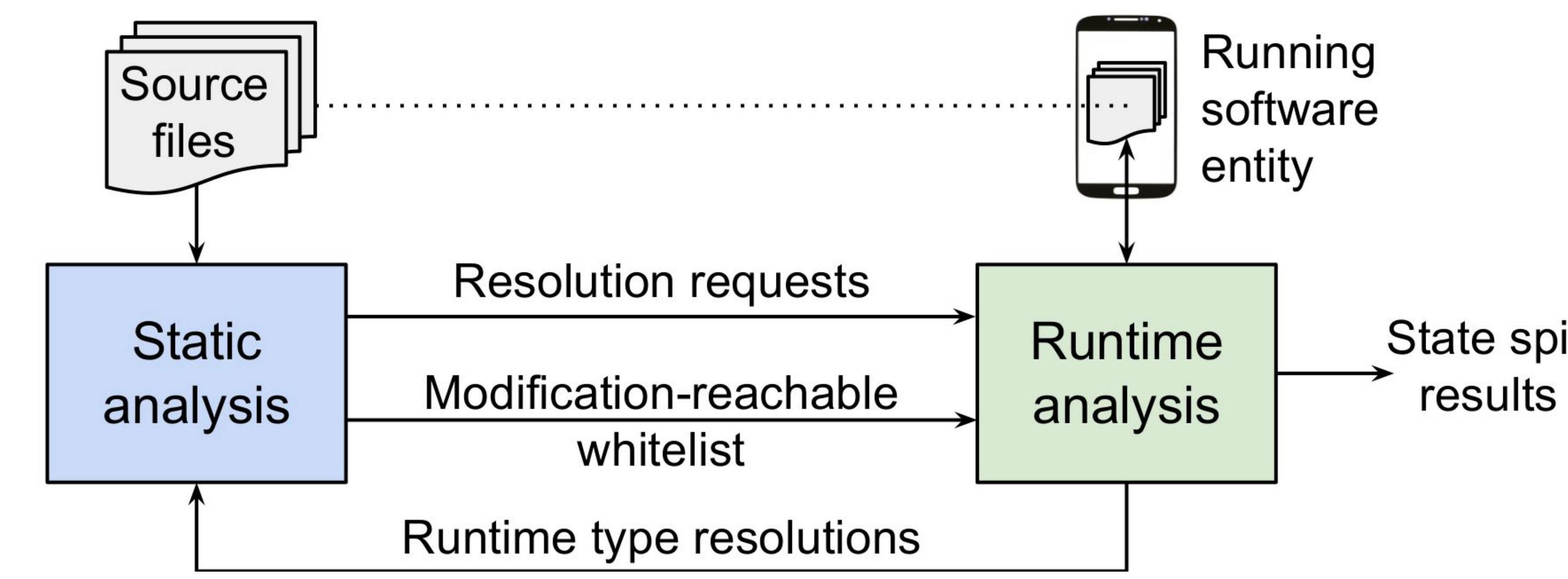


Designs to avoid state spill

- Client-provided resources
- Stateless communication
- Hardening of entity state
- Modularity without interdependence
- Separation of multiplexing from indirection

RESTful principles

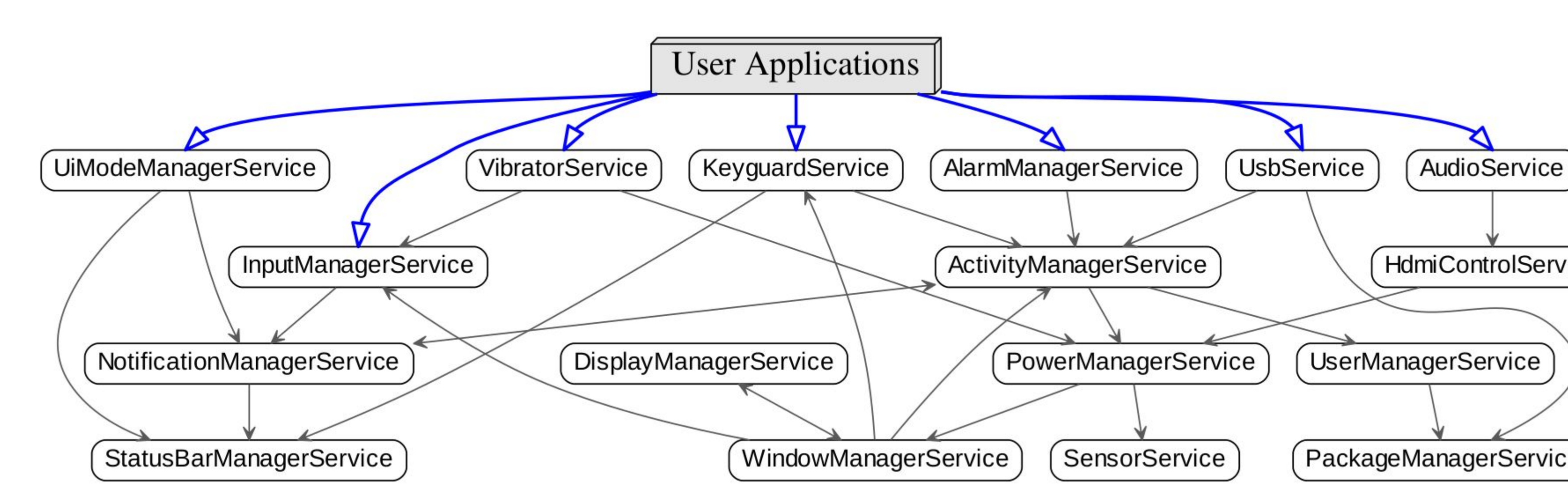
Automated detection with StateSpy



1) Detect *quiescent point* for safe analysis
   -- monitor transaction entry & exit points
2) Capture state of software entity
   -- key insight: use **debugging frameworks**
3) Difference captured states
   -- via existing tree comparison algorithms
4) Filter results with static analysis
   -- determine *modification reachability*

State spill in Android system services

- StateSpy found state spill in 94% of Android services analyzed, most with 1-10 instances
- Classified state spill instances in 60 transactions:
  - 39% caused by indirection layers
  - 21% caused by multiplexers
  - 55% from dispatchers/collaboration
- Better discovery of problems in app migration than manual identification of residual dependencies [1]
- Discovered *secondary spill* in 27 services:



[1] Alex Van't Hof, et al., *Flux: Multi-Surface Computing in Android*, EuroSys'15.