Outline

• Discussion of the 3 assigned papers
  ➢ TTD07: High Performance Java Sockets for Parallel Computing on Clusters
  ➢ RPS06: Corona: A High Performance Publish-Subscribe System for the World Wide Web
  ➢ ESB06: Publish-subscribe for High-performance Computing
• Tailored Java sockets library for parallel applications on clusters
• Can act as a transport layer for Java RMI
• Java communication middleware issues
  ➢ Lack of support for high-speed interconnects
  ➢ Need for TCP/IP stack emulations over high-speed interconnects
  ➢ Need for efficient protocols for cluster communications

• Java Sockets for parallel computing (JFS)
  ➢ High performance on high speed networks
    ❑ Java Native Interface (JNI) access to high performance native communication libraries
  ➢ JFS Communication performance increase
    ❑ Existing implementations: buffering and unnecessary copies → One-copy protocol and zero-copy protocol
  ➢ Serialization cost reduction
    ❑ Native method to process arrays of primitive data types and transform to byte arrays
Figure 1. Default Java sockets communication

Figure 2. One-copy JFS communication
• Transparent to user
• No source code modification
• Use of native implementations through JNI if available

Listing 1. Replacing default Sockets by JFS

```java
SocketImplFactory factory = new JFSFactory();
Socket.setSocketImplFactory(factory);
ServerSocket.setSocketFactory(factory);
Class cl = Class.forName(className);
Method method = cl.getMethod("main", argsTypes);
method.invoke(null, parameters);
```
Java RMI optimization

- Methods overhead analysis
  - Network (84%), RMI Protocol processing, mainly stub and skeleton operation (3.3%), Serialization (12.7%), and DGC (0.2%)

- Assumptions
  - The use of a shared file system from which the classes can be loaded
  - Homogeneous architecture of the cluster nodes
  - The use of a single JVM version

Java RMI optimization ➔ Performance improvement

- RMI Data Transport Management
  - Managing data to reduce buffering and sockets delays

- Serialization Overhead Reduction
  - JFS native serialization

- Protocol Overhead Reduction
  - Assumptions hold true to reduce overhead
• Frequently updated content, such as Weblogs, collaboratively authored Web pages (wikis), and news sites, motivates a 
  publish-subscribe mechanism

• asynchronous update notification
  
  ➢ *micronews syndication* tools based on naive, repeated polling (RSS for example) ➢ poor performance and scalability

  ➢ From Wikipedia

  □ “web syndication refers to making web feeds available from a site in order to provide other people with a summary of the website's recently added content”
• A novel, decentralized system for detecting and disseminating Web page updates (Corona: Cornell Online News Aggregator)
• Determine the optimal resources to devote to polling data sources in order to meet system-wide goals
• Fundamental tradeoff between bandwidth and update latency
  ➢ Optimization problem
  ➢ computes the optimal way to allocate bandwidth to monitored Web objects using a decentralized algorithm that works on top of a distributed peer-to-peer overlay
  ❏ Polling schedule

• Topic-based publish-subscribe system for the Web
• URLs of Web content serve as topics or *channels* in Corona
• users register their interest in some Web content by providing its URL and receive updates asynchronously about changes posted to that URL
• Any Web object identifiable by a URL can be monitored with Corona
• Corona checks for updates on registered channels by cooperatively polling the content servers from geographically distributed nodes
Figure 1: Corona Architecture: Corona is a distributed publish-subscribe system for the Web. It detects Web updates by polling cooperatively and notifies clients through instant messaging.

Figure 2: Cooperative Polling in Corona: Each channel is assigned a wedge of nodes to poll the content servers and detect updates. Corona determines the optimal wedge size for each channel through analysis of the global performance-overhead tradeoff.
• *Corona-Lite*: minimizing average update detection time while bounding the total network load placed on content servers

• *Corona-Fast*: minimizes the total network load on the content servers while meeting a target average update detection time

• *Corona-Fair*: incorporates the update rate of channels into the performance tradeoff. Consider the ratio of the update detection time and the update interval of the channel, which it minimizes to achieve a target load

• *Corona-Fair-Sqrt*

• *Corona-Fair-Log*

  ➔ A sub-linear metric dampens the tendency of the optimization algorithm to punish slow-changing yet popular feeds.

• *Echo*: High-performance event-delivery middleware to scale to data rates found in grid environments

• Channel (event channel) and type-based publish-subscribe systems (event sinks and sources)

• The process that creates the event channel is the contact point for other processes wishing to use the channel

• A source sends event messages directly to all sinks, and network traffic for individual channels is multiplexed over shared communication links

• *Echo* is implemented on top of Portable Binary I/O (PBIO) package
• Supports null-terminated strings and dynamically sized arrays
• Combining native data representation (NDR) with dynamic generation of unmarshalling routines (sender NDR is the wire format)
• Each marshalled data package contains a format cookie that identifies the meta-information associated with the data
  ➢ Intervening hosts can filter or transform events without a priori knowledge of the event’s contents
• Allowing variation in the data types associated with a single channel
Derived event channels

- Allow general computations over event data and ensures their efficient execution through
  - use of dynamic code generation and decentralized event dist.
- Applications that want to customize their event data can create a new channel
- Contents are derived from those of a preexisting channel through an application-provided derivation function, $F$
- The event channel implementation moves $F$ to all event sources in the original channel
  - executes it locally whenever events are submitted
  - transmits any resulting event(s) via the derived channel

Echo's derived event channel relies on E-Code (subset of C)

- $F$ expressed in E-Code and DCG used to create a native version of $F$ on the source
- Simple mode: the derived channel’s event type is the same as that of the original channel
- Could associate sources providing unfiltered events to derived event channels
- Complex mode: event type associated with the derived channel is different from that of parent channel
- See example in Figure 3
• Earlier implementations: point-to-point messaging with direct source-to-sink event delivery
• Echo on Overlays (EVPath)
  ➢ EVPath: middleware package designed to facilitate the dynamic composition of overlay networks for message passing
• Stepping stones compose a data path
  ➢ Stones have associated queues
• Output stones, terminal stones, split stones, and filter stones
• Ability to freeze portions of message flow
• Higher software layers perform stone management

Figure 4. Changes in an overlay network. (a) An overlay with localized congestion or computational overload, and (b) the same overlay with additional processing stones.