Programación de Sistemas y Concurrencia

Tema 7: Introducción a la Programación Dirigida por Eventos

Grado en Ingeniería Informática

Grado en Ingeniería del Software

Grado en Ingeniería de Computadores

Índice

- Eventos vs. Concurrencia
- Eventos y manejadores
- Colas de eventos
- Patrones de interacción basados en eventos y Marcos de Trabajo
- GUIs avanzadas. Concurrencia y eventos en GUIs

Eventos vs. Concurrencia

- En los sistemas reactivos hay que tener en cuenta otros factores además de la concurrencia:
  - Escalabilidad, rendimiento, …
- Cuando la "carga" del sistema aumenta se debería proporcionar una degradación "suave" del rendimiento
  - Ej: servicios de Internet con miles de usuarios
- Los eventos son una alternativa a la concurrencia/hebras para la implementación de sistemas reactivos
  - También pueden combinarse ambas aproximaciones

Eventos vs. Concurrencia

- Modelo de hebras
  - Un proceso contiene múltiples hebras, cada una de las cuales puede ser planificada o bloqueada en cualquier momento.
- Modelo de eventos
  - Un proceso consiste en el procesamiento de un conjunto de eventos. En cualquier momento **sólo** se está procesando un evento.

Eventos vs. Concurrencia

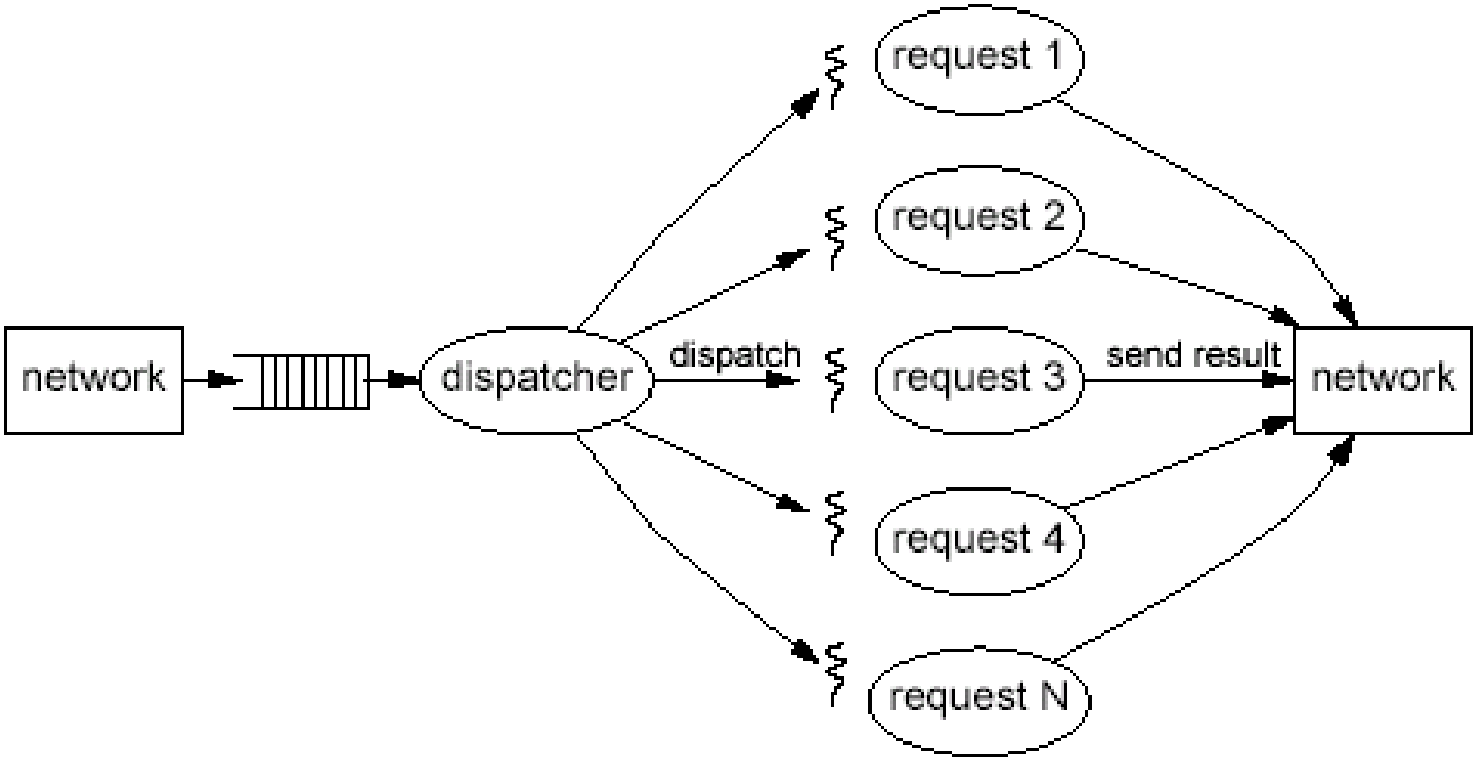
- ¿Qué es un evento?
  - Un objeto “encolado” preparado para ser utilizado por algún módulo (posiblemente en orden secuencial)
- Los eventos pueden ser definidos y tratados a diferentes niveles
  - Por el sistema operativo
  - Por un lenguaje de programación
  - Por un framework
- El flujo de los eventos dirige al programa
- Los eventos son procesados en los **manejadores de eventos** (handlers)

Eventos vs. Concurrencia

- Usos típicos de los eventos
  - Principalmente en interfaces de usuario (GUIs)
    - Un manejador para cada evento (clic en botón)
    - Los manejadores implementan el comportamiento (deshacer, borrar fichero, etc.)
  - Sistemas distribuidos
    - Un manejador para cada fuente de datos (ej: socket)
    - Los manejadores procesan las solicitudes de entrada y envían las respuestas
    - E/S dirigida por eventos para la superposición de E/S
  - Incluso para la comunicación de módulos o componentes software

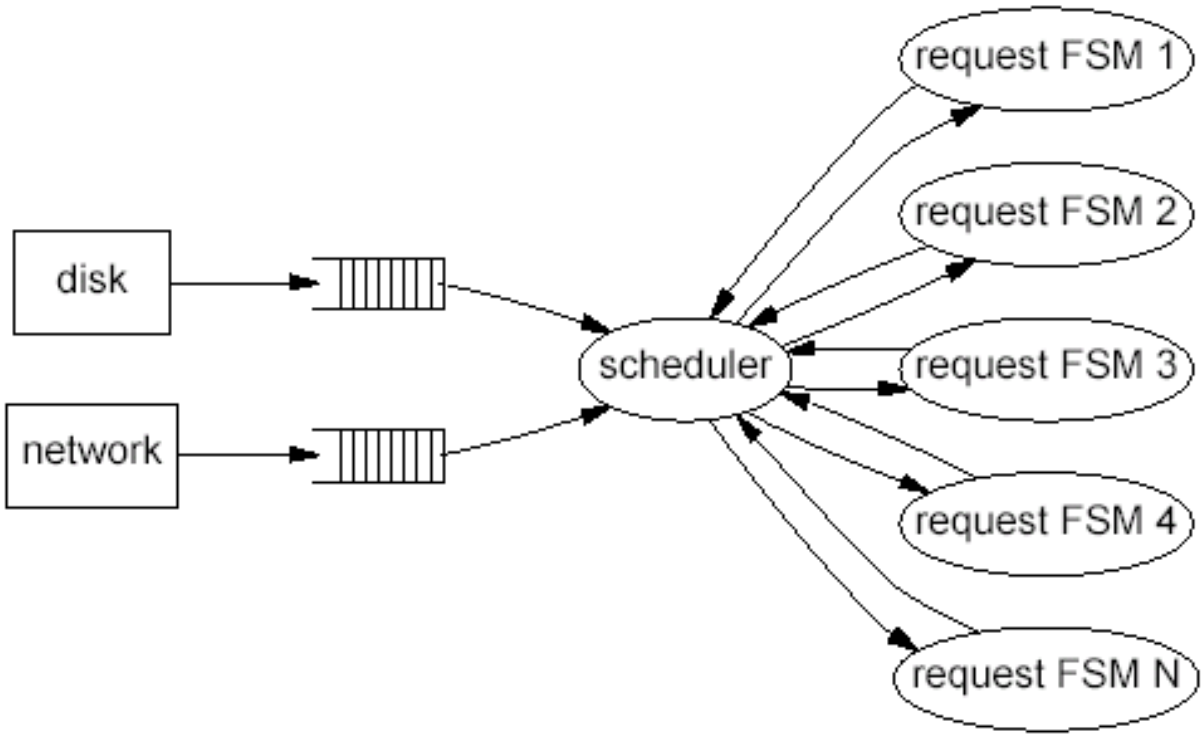
Eventos vs. Concurrencia

Modelo basado en hebras



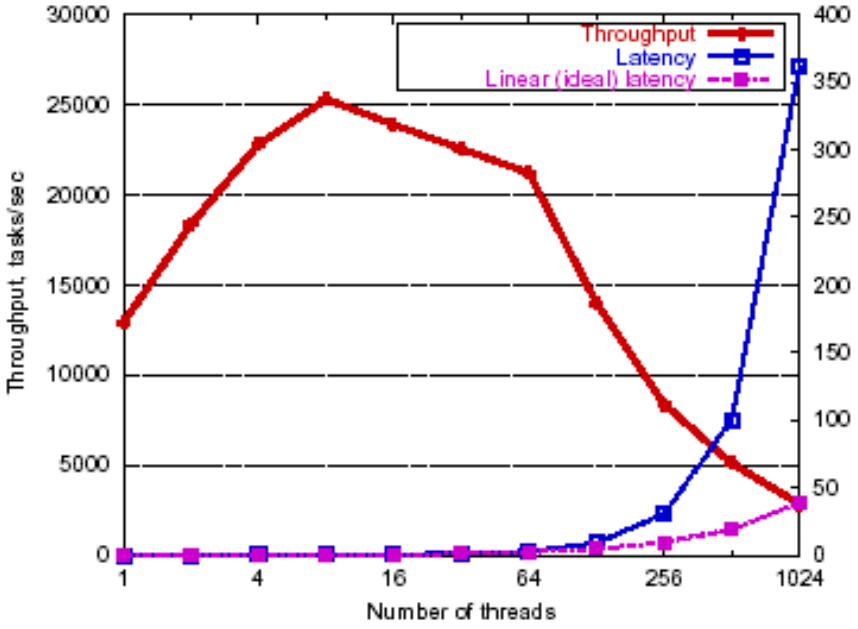
Eventos vs. Concurrencia

Modelo basado en eventos

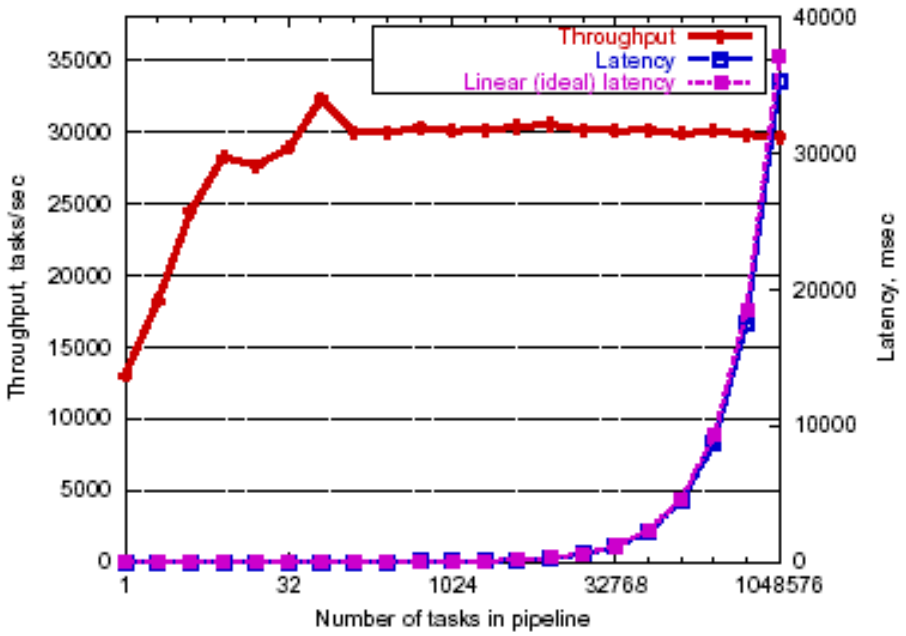


Eventos vs. Concurrencia

Servidor con hebras

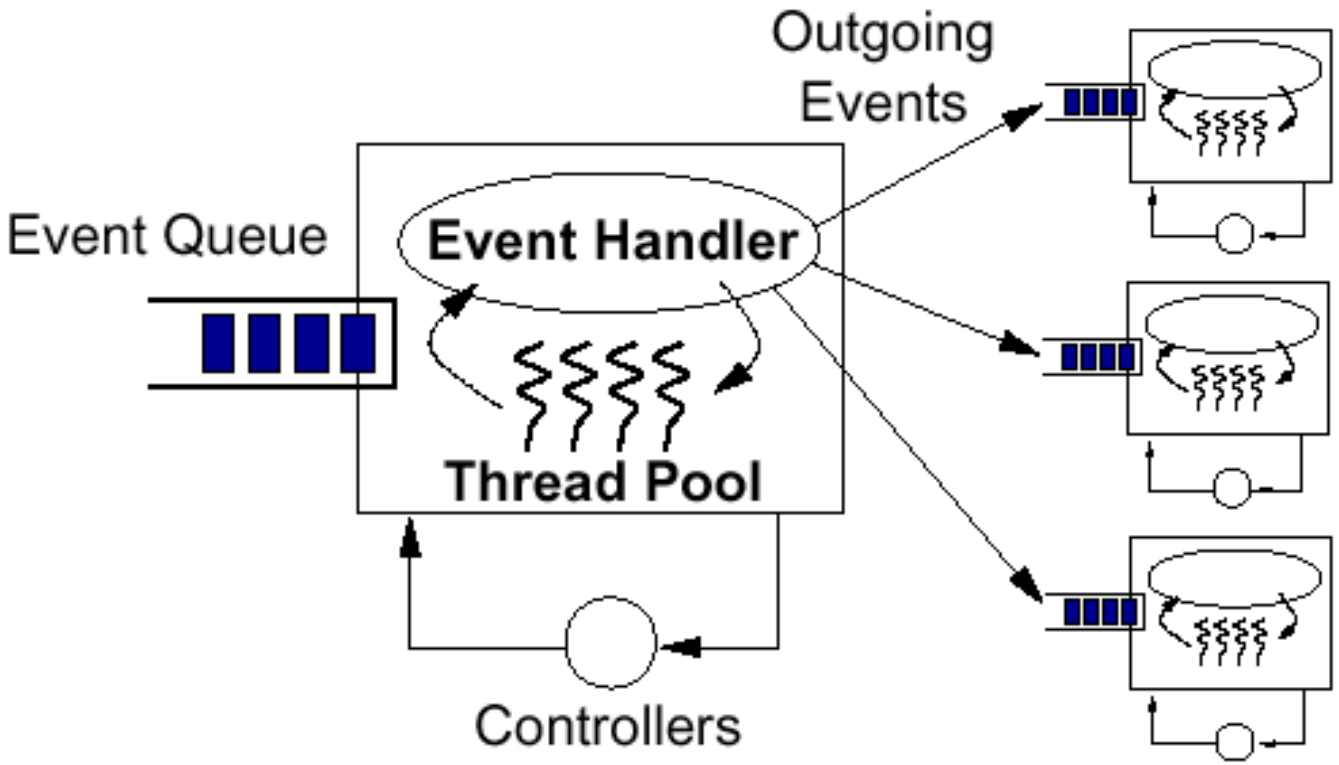


Servidor con eventos



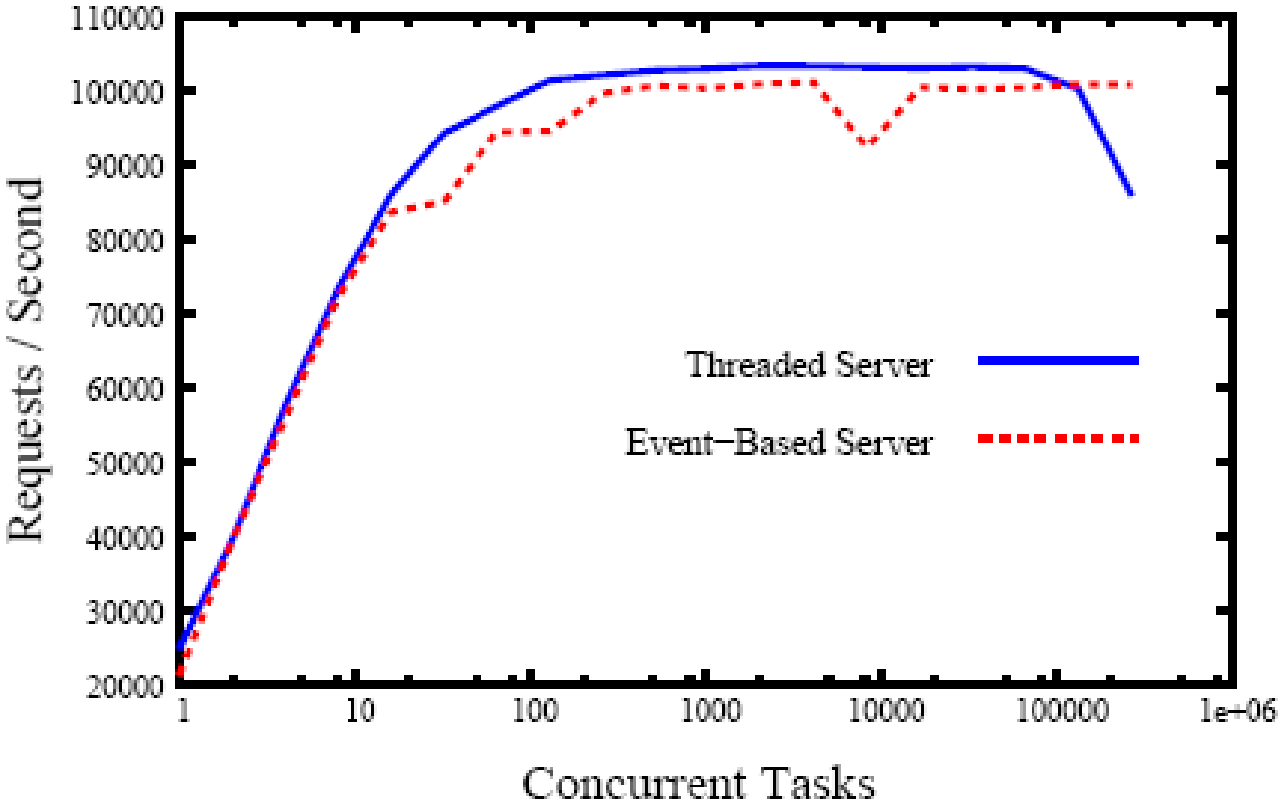
Eventos vs. Concurrencia

Un sistema mixto



Eventos vs. Concurrencia

Servidor con hebras utilizando thread pools



Eventos vs. Concurrencia

Eventos

- Manejadores de eventos
- Eventos aceptados por un manejador
- SendMessage/AwaitReply
- SendReply
- Esperando por mensajes

Hebras

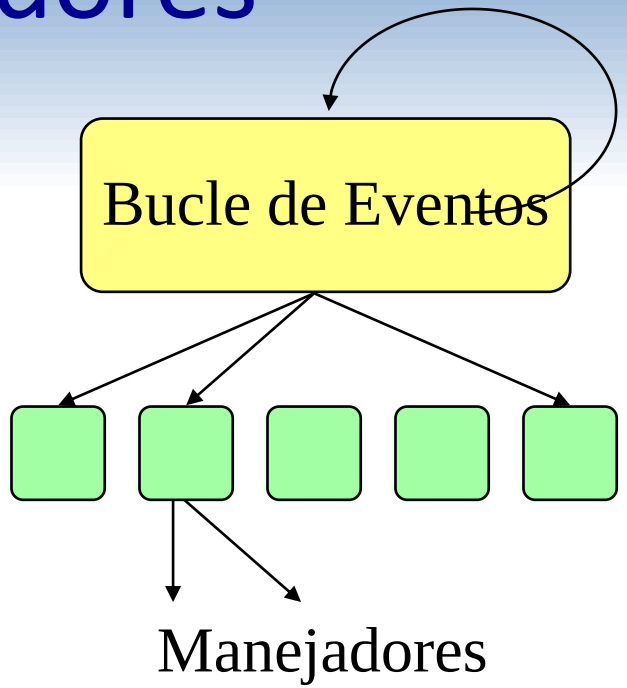
- Monitores
- Funciones exportadas por un módulo
- Llamada a procedimiento o fork/join
- Retorno del procedimiento
- Esperando en variables de condición

Eventos vs. Concurrencia

- Ambas aproximaciones (hebras/eventos) tienen ventajas e inconvenientes
  - Facilidad de programación con hebras
    - Dificultad de la sincronización
  - Rendimiento con eventos
    - Pérdida del flujo de control
- Usar uno u otro modelo cuando esté justificado
  - Ej: hebras cuando la concurrencia está justificada

Eventos y manejadores

- Un único flujo de ejecución: no hay concurrencia
- Se indica el interés en determinados eventos
  - Subscripciones o callbacks
- El “bucle de eventos” espera por los eventos e invoca a los manejadores
- Los manejadores de eventos *no son* interrumpibles
  - Sí por el sistema (recolector de basura, …)
- Los manejadores suelen tener poco código

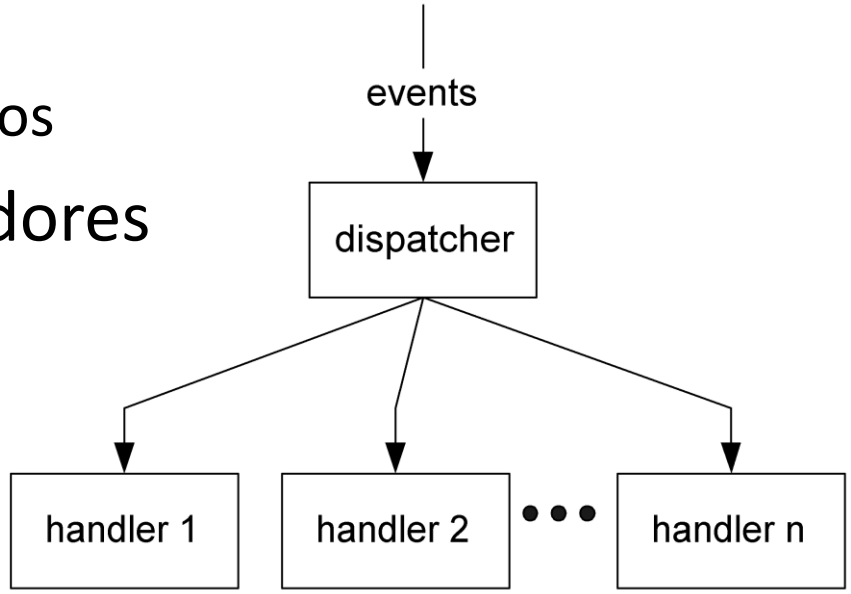


Eventos y manejadores

- Características de los manejadores de eventos
  - No pueden bloquearse → no hay sincronización
  - No deberían compartir memoria
    - Al menos no en paralelo
  - Toda la comunicación se realiza a través de eventos

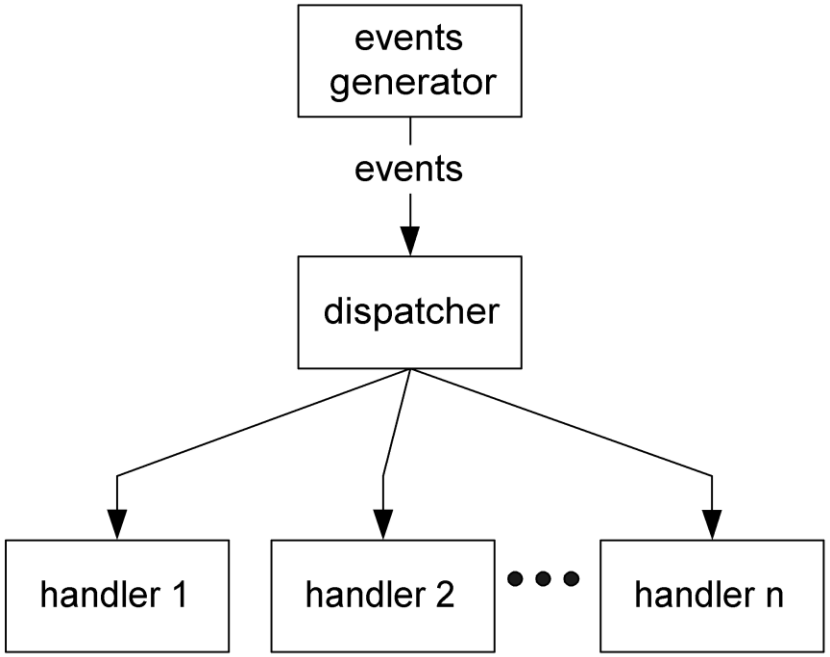
Eventos y manejadores
El patrón de diseño manejador

- El tratamiento de eventos en un sistema puede verse como un patrón de diseño
  - Flujo de datos de entrada: *eventos*
  - Un dispatcher
    - Incluye al bucle de eventos
  - Un conjunto de manejadores



Eventos y manejadores
El patrón de diseño manejador

- Patrón de diseño con generador de eventos
  - Se incluye un posible generador de eventos (hardware o software)



Eventos y manejadores

- En aplicaciones hardware
  - Flujo de eventos “infinito”
- Habitualmente, flujo finito con evento final especial
  - Botón close
  - Tecla ESC
  - Fin de fichero o flujo de entrada
  - …
- No siempre hay que tratar todos los eventos
  - No es necesario tener un manejador para cada evento
  - El sistema “traslada” o “descarta” el evento de manera similar al tratamiento de excepciones

Eventos y manejadores

- Bucle de eventos típico incluido en dispatcher

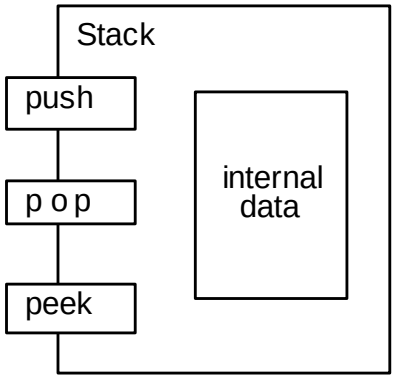
```
do {
    Obtener evento de la lista de eventos
    switch (tipo_evento) {
        case event_type1: handler1(args);
                          break;
        case event_type2: handler2(args);
                          break;
        ...
        case event_end: quit();
                        break;
        default: ignorar o elevar al nivel superior
    }
}
```

Colas de eventos

- En ocasiones, los eventos se generan a una velocidad mayor de la que pueden ser tratados
  - El dispatcher puede estar ocupado y no tratar el evento
  - Solución: utilizar una cola de eventos
- Las aplicaciones GUI típicamente utilizan una cola de eventos
- Cada vez que el dispatcher termina de tratar un evento saca al siguiente evento de la cola
- Las colas de eventos también pueden utilizarse de manera explícita por el programador
  - Modelos publish/subscribe

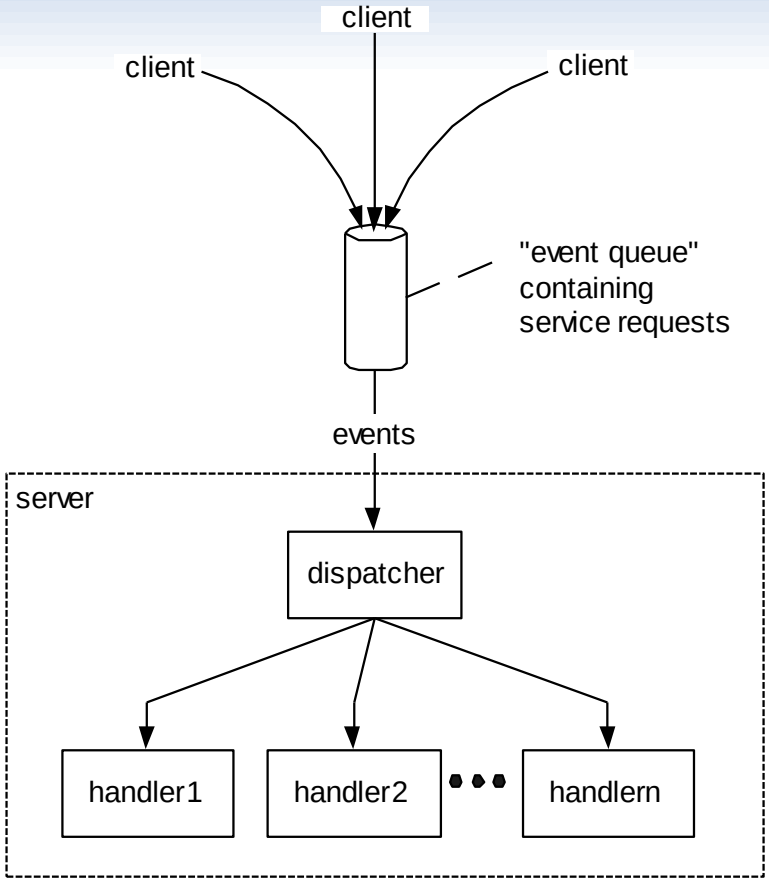
Patrones de interacción y marcos de trabajo

- El modelo de programación con eventos aparece de forma implícita/explícita en muchas aproximaciones
- POO
  - Métodos como manejadores de eventos
    - Sin dispatcher



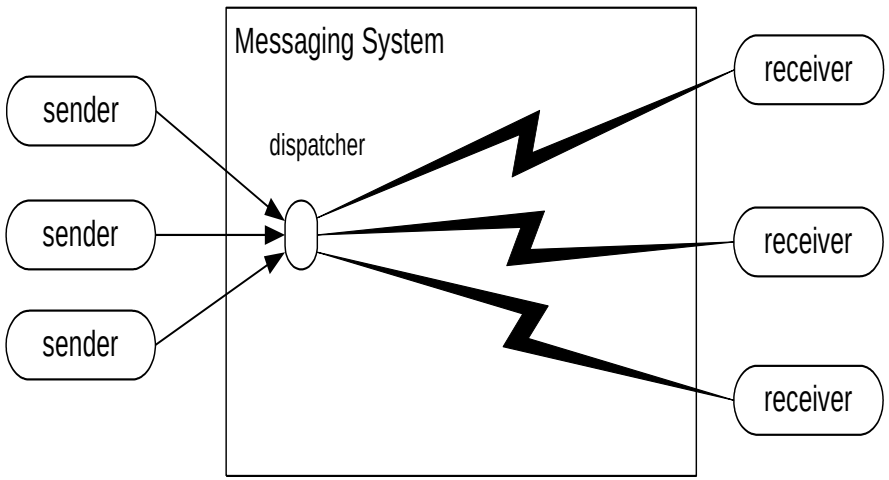
Patrones de interacción y marcos de trabajo

- Arquitectura cliente-servidor
  - El servidor (hardware o software) proporciona *servicios* a los clientes que usan el servicio
  - Ej: servidores de impresión, servidores web, …



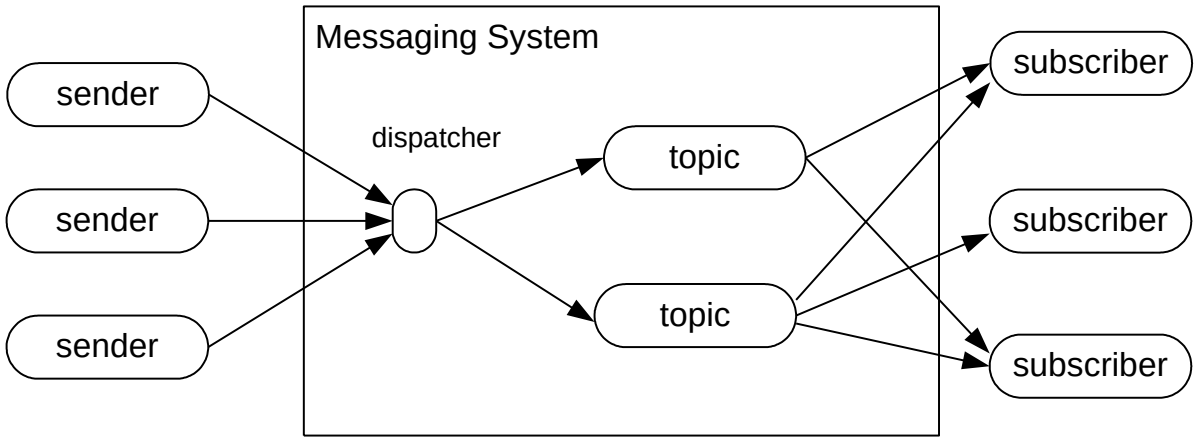
Patrones de interacción y marcos de trabajo

- Sistemas de mensajes
  - Obtener en los manejadores de eventos (*receptores*) *eventos* (mensajes) de los generadores de eventos (*emisores*)
  - Posible localización física o plataforma diferentes para emisor y receptor



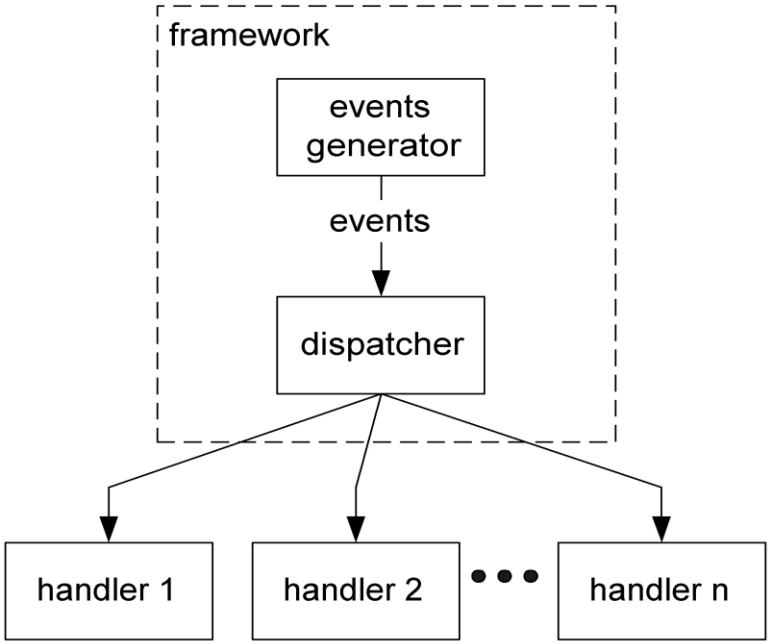
Patrones de interacción y marcos de trabajo

- Message-oriented middleware (MOM)
  - Caso particular de sistemas de mensajes
  - Comunicación entre diferentes aplicaciones
  - Habitualmente utiliza el modelo *publish/subscribe*



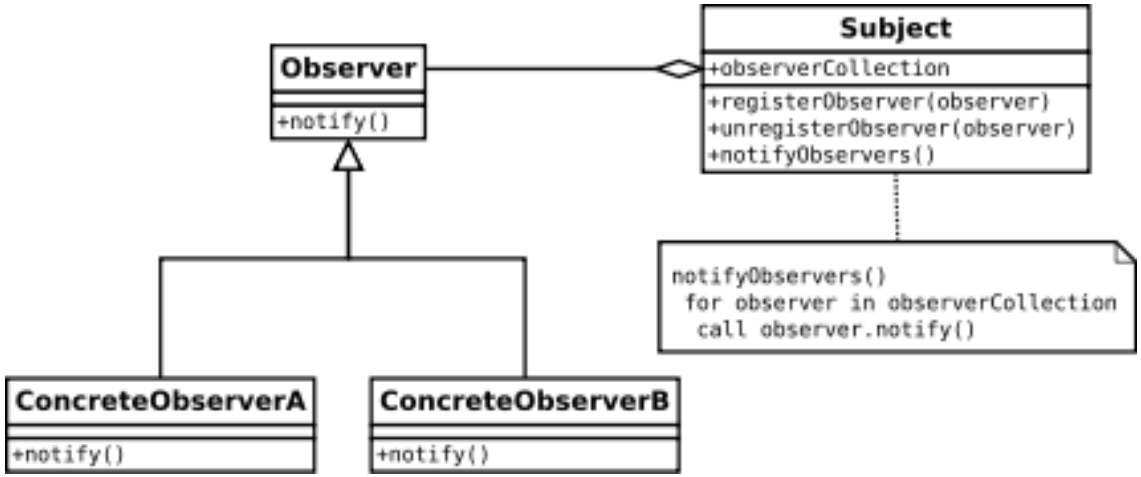
Patrones de interacción y marcos de trabajo

- Marcos de trabajo (frameworks)
  - *Pieza* de software que define puntos de conexión que deben ser conectados para ser utilizado
  - Utilización implícita del patrón *Manejador*
  - Se “pierde” el flujo de control. El framework “llama” al código, al contrario que en las librerías que son “llamadas” por el programador
  - Muchas GUIs están basadas en el uso de marcos de trabajo



Patrones de interacción y marcos de trabajo

- El patrón *Observador*
  - Ampliamente utilizado en programación con eventos para GUIs
  - Existe una entidad *Sujeto* (*subject)* y múltiples *observadores*
  - El Sujeto proporciona mecanismos para que los observadores se **registren** en temas de su interés (**objetos de cierto tipo**) y sean notificados cuando ocurran eventos sobre esos temas
  - También conocido como patrón *publish/subscribe*
    - *Se envían y reciben los objetos del tipo de evento con sus atributos particulares*
  - Es un caso especial del patrón *Manejador*



Patrones de interacción y marcos de trabajo

- Modelo Vista-Controlador (MVC)
  - Basado en el patrón Observador
  - El *Modelo* es un objeto que gestiona los datos y comportamiento de cierto dominio de aplicación
  - Las *vistas* se registran con el modelo como observadores
  - Cuando el *controlador* realiza algún cambio al modelo, este notifica a los observadores (las *vistas*) que el modelo ha cambiado
  - Dos versiones
    - Pull
      - El objeto evento (modelo) no contiene prácticamente ninguna información, sólo se avisa del cambio. La vista consulta al modelo (pull)
    - Push
      - El modelo pone (push) la información cambiada en las vistas

Concurrencia y eventos en GUIs

- La programación de GUIs no es una tarea trivial
  - Hay que definir la apariencia y comportamiento de cada control
  - Existen *muchas* clases de eventos que deben ser manejadas por la GUI
    - Eventos hardware y software
- Muchos sistemas actuales de GUI están implementados con una única hebra
  - Swing, SWT, Qt, NextStep, MacOS, Cocoa, X Windows, …
    - AWT soportaba un mayor grado de concurrencia
  - Por numerosos problemas con interbloqueos
  - Los eventos son tratados de manera **secuencial**

Concurrencia y eventos en GUIs

- La programación de GUIs tiene sus *peculiaridades* con las hebras
  - "Ciertas" tareas deben realizarse en la hebra del manejador de eventos
  - No se *pueden* realizar tareas de larga duración en los manejadores de eventos
    - Para no "bloquear" la interfaz
  - Las estructuras de datos no suelen ser *thread-safe*
    - El comportamiento "seguro" se consigue confinando la utilización de todos los objetos de la GUI en la hebra de eventos
- Muchos sistemas de GUIs están basados en frameworks
  - Suministran el bucle de eventos y la cola de eventos

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- Java proporciona mecanismos para trabajar con eventos con AWT y Swing.
- El paquete java.awt.event contiene los tipos de eventos

ActionEvent	InvocationEvent
AdjustmentEvent	ItemEvent
ComponentEvent	KeyEvent
ContainerEvent	MouseEvent
FocusEvent	MouseWheelEvent
InputEvent	PaintEvent
InputMethodEvent	TextEvent

- Cada tipo de evento contiene atributos y métodos apropiados para ese tipo de eventos
  - Ej: MouseEvent
    - MouseEvent.*getButton()*
    - MouseEvent.*getClickCount()*
    - MouseEvent.*getPoint()*
    - KeyEvent.*getKeyChar()*

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- En java.awt.event se encuentran también distintos tipos de Listeners (interfaces) para los distintos tipos de eventos, todos los cuales heredan de la interfaz java.util.EventListener

ActionListener	MouseListener
ContainerListener	MouseMotionListener
FocusListener	MouseWheelListener
InputMethodListener	TextListener
ItemListener	WindowFocusListener
KeyListener	WindowListener

- Una interfaz *listener* tiene una colección de métodos (manejadores de eventos) cada uno de los cuales maneja el mismo tipo de eventos
  - *Ej: MouseListener* maneja eventos del ratón
    - mouseClicked(MouseEvent e)
    - mouseEntered(MouseEvent e)
    - mouseExited(MouseEvent e)
    - mousePressed(MouseEvent e)
    - mouseReleased(MouseEvent e)

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- En Java los componentes gráficos actúan como las entidades *Subject* del patrón observador
  - Ofrecen eventos a los que subscribirse mediante métodos *add[xxx]Listener*
    - addMouseListener(), addActionListernet(), …
    - Estos métodos están definidos vía herencia (ej: java.awt.Component, javax.swing.AbstractButton)
  - Cuando un evento ocurre sobre un componente, éste, llama al método correspondiente del objeto listener registrado y le pasa el objeto evento
- Ej:

```
final Random random = new Random();
final JButton button = new JButton("Cambiar color");
...
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button.setBackground(new Color(random.nextInt()));
    }
});
```

- Si no quisiéramos implementar todos los métodos de la interfaz Listener, Java proporciona también **clases adaptadoras** que proporcionan implementaciones por defecto para todos los métodos de una intefaz. Ej: MouseAdapter

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- En Swing podemos distinguir 3 tipos de hebras
  - Hebras iniciales encargadas de ejecutar el código inicial
  - La hebra *manejadora de eventos*
    - Para saber si estamos ejecutando la hebra de eventos podemos utilizar:
      - javax.swing.SwingUtilities.isEventDispatchThread
  - Hebras trabajadoras para tareas que consuman tiempo
    - Swing proporciona la clase javax.swing.SwingWorker para simplificar la tarea de usar hebras trabajadoras
- Una hebra inicial en Swing típicamente crea un objeto Runnable que inicializa la GUI y planifica al objeto para su ejecución en la hebra de eventos

```
SwingUtilities.invokeLater(new Runnable() {  // o también invokeAndWait
    public void run() {
        createAndShowGUI();      // Inicializa componentes gráficos
    }
});
```

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- SwingWorker es una clase abstracta (incluida en Java 6)
  - Debemos definir una subclase o utilizar clases anónimas para crear objetos de tipo SwingWorker
  - Método **execute** para comenzar la tarea, **get** para esperar y obtener resultados
  - Define un método **done**, automáticamente invocado por la hebra de eventos cuando la tarea finaliza
  - Implementa java.util.concurrent.Future. Permite proporcionar un valor de retorno a otra hebra (o cancelación)
  - Permite obtener resultados intermedios con el método **publish** que invoca automáticamente a **process** desde la hebra de eventos
  - Permite utilizar propiedades para comprobar el estado actual de la hebra trabajadora

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- Tarea en segundo plano simple (carga de imágenes)

```
SwingWorker worker = new SwingWorker<ImageIcon[], Void>() {
   @Override
   public ImageIcon[] doInBackground() {
      final ImageIcon[] innerImgs = new ImageIcon[nimgs];
      for (int i = 0; i < nimgs; i++) {
         innerImgs[i] = loadImage(i+1);
      }
      return innerImgs;
   }
```

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- Tarea en segundo plano simple (carga de imágenes) (continuación)

```
@Override
   public void done() {
      //Eliminar la etiqueta "Loading images".
      animator.removeAll();
      loopslot = -1;
      try {
         imgs = get();
      } catch (InterruptedException ignore) {}
      catch (java.util.concurrent.ExecutionException e) {
            …
         System.err.println("Error recuperando imagen");
      }
   }
};
```

Faltaría lanzar la tarea con worker.execute();

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- Todas las subclases concretas de SwingWorker implementan **doInBackground**, el método **done** es opcional
- SwingWorker es una clase genérica con dos parámetros
  - El primer tipo especifica el tipo de retorno para doInBackground y también para el método **get** (usado por otras hebras para recuperar el valor retornado)
  - El segundo parámetro especifica un tipo para resultados intermedios
    - Void si no se utiliza
- ¿Por qué se ha hecho de esta forma?
  - ¿No habría bastado con hacer la carga de imágenes en doInBackground?

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- Tareas con resultados intermedios
  - Utilización de los métodos **publish** y **process** (invocado desde la hebra de eventos)

```
private class FlipTask extends SwingWorker<Void, FlipPair> {
…
@Override
protected Void doInBackground() {
   long heads = 0;
   long total = 0;
   Random random = new Random();
   while (!isCancelled()) {
      total++;
      if (random.nextBoolean()) {
         heads++;
      }
      publish(new FlipPair(heads, total));
   }
   return null;
}
```

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- Tareas con resultados intermedios (continuación)
  - Cuando **process** es invocado, pueden haberse acumulados muchos datos

```
protected void process(List<FlipPair> pairs) {
    FlipPair pair = pairs.get(pairs.size() - 1);
        …
}
```

- Cancelación de tareas
  - Para cancelar una tarea se puede invocar SwingWorker.cancel
  - La tarea debe cooperar para ser cancelada
    - Terminando cuando recibe una interración
    - O utilizando SwingWorker.isCancelled periódicamente (ver ejemplo anterior)

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- Consulta del estado de hebras trabajadoras
  - Se dispone de las propiedades **progress** y **state**
  - Implementando un listener para el cambio de estas propiedades es posible monitorizar el grado de progreso de la tarea y su estado
  - La propiedad progress es un valor entero en el rango 0..100
    - Métodos SwingWorker.setProgress y SwingWorker.getProgress
  - La propiedad state puede tomar los valores
    - PENDING: durante la consttucción del objeto y justo antes de ser invocado doInBackground
    - STARTED: desde la ejecución de doInBackground hasta justo antes de ser invocado done
    - DONE: finalizado
    - El método SwingWorker.getState permite consultar el estado de un objeto

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- Ejemplo barra de progreso

```
class Task extends SwingWorker<Void, Void> {
    public Void doInBackground() {
        Random random = new Random();
        int progress = 0;
        setProgress(0);
        while (progress < 100) {
            try {
            Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException ignore) {}
            progress += random.nextInt(10);
            setProgress(Math.min(progress, 100));
        }
    return null;
    }
```

Concurrencia y eventos en GUIs

Eventos y GUIs en Java

- Ejemplo barra de progreso (continuación)
  - Creación de la tarea

```
task = new Task();
task.addPropertyChangeListener(this);
task.execute();
```

  - Implementación del listener

```
public void propertyChange(PropertyChangeEvent evt) {
  if ("progress" == evt.getPropertyName()) {
      int progress = (Integer) evt.getNewValue();
      progressBar.setValue(progress);
      }
}
```

Referencias

- Event-Driven Programming: Introduction, Tutorial, History. Stephen Ferg.
  http://creativecommons.org/licences/by/2.5/
- Why Events Are A Bad Idea (for high concurrency servers). R.v. Behren, J. Condit, E. Brewer. Proceedings of the HotOS IX: the 9th Workshop on Hot Topics in Operating Systems, 2003
- SEDA: An Architecture for well-conditioned scalable internet services. M. Welsh, D. Culler, E.Brewer, R. Agarwal. Proceedings of Eighteeth Symposium on Operating Systems Principles, 2001
- Concurrency in Swing. The Java™ Tutorials. http://docs.oracle.com/javase/tutorial/uiswing/concurrency/