The new Gtk+ Printing API

Alexander Larsson – alexl@redhat.com

John (J5) Palmieri – johnp@redhat.com

Why printing support in Gtk+

- Want to make Gtk+ a “full” toolkit
- Cairo drawing model makes this easy
- libgnomeprint unmaintained
- Project Ridley

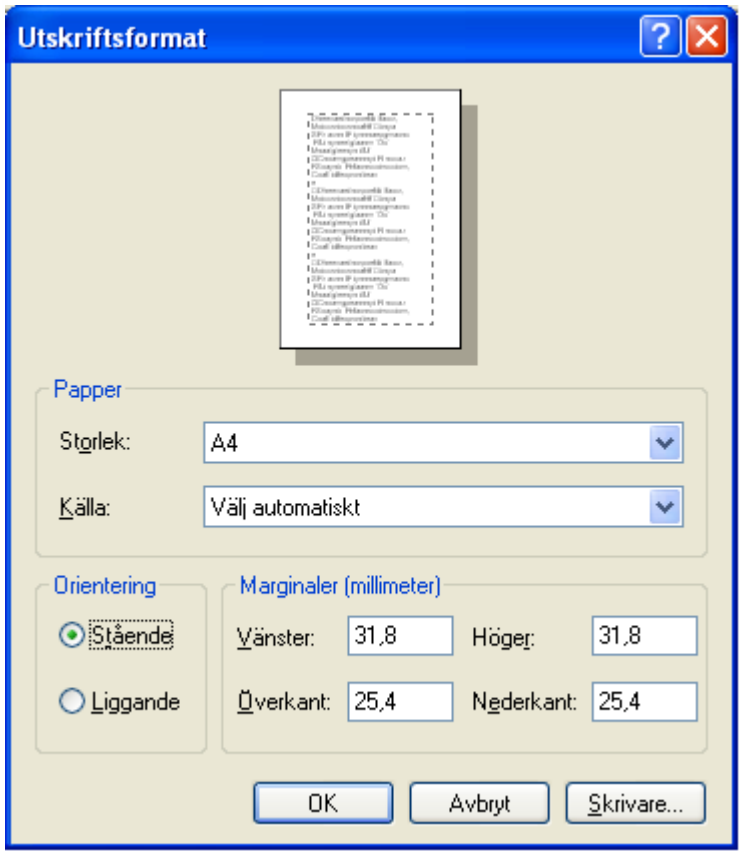
A look at other printing dialogs

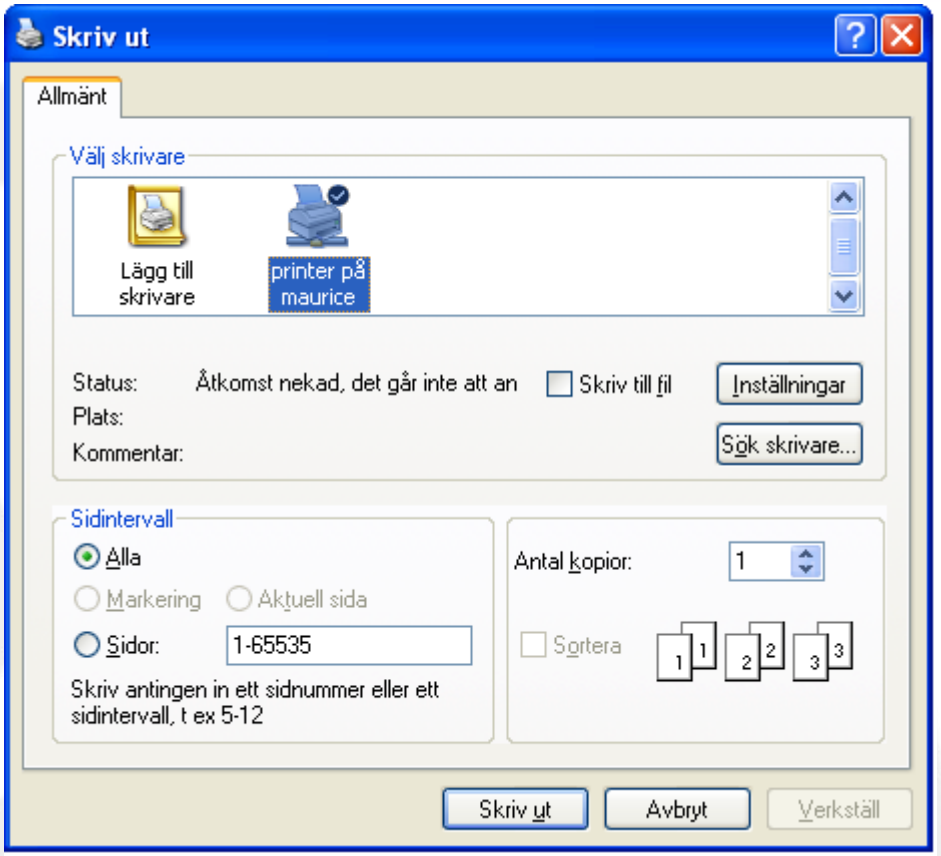
- Win32
- OSX
- GnomePrint
- KDE

More dialogs at:

`http://people.redhat.com/~alexl/print_dialogs/`

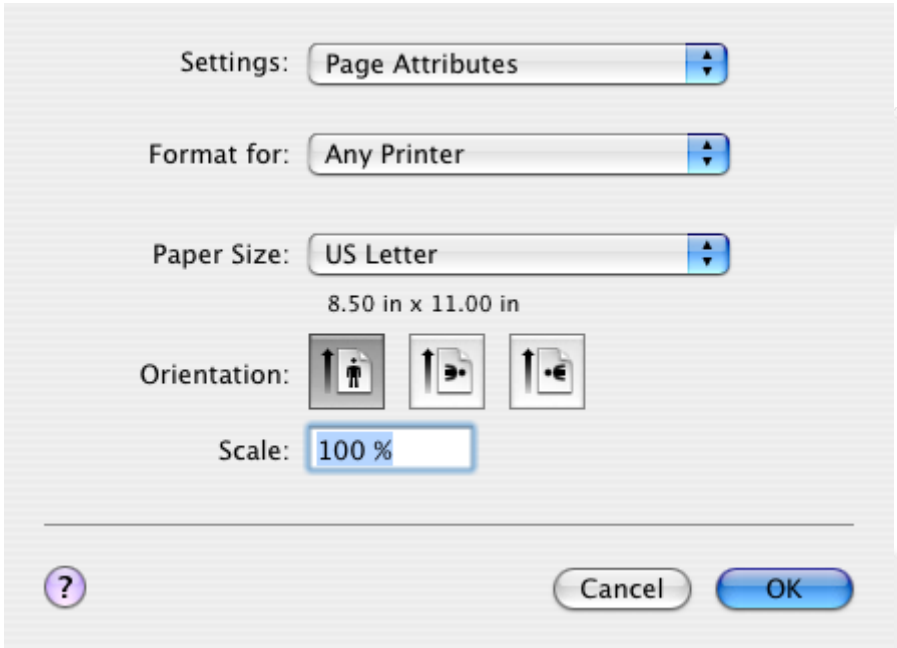
Win32 print dialogs

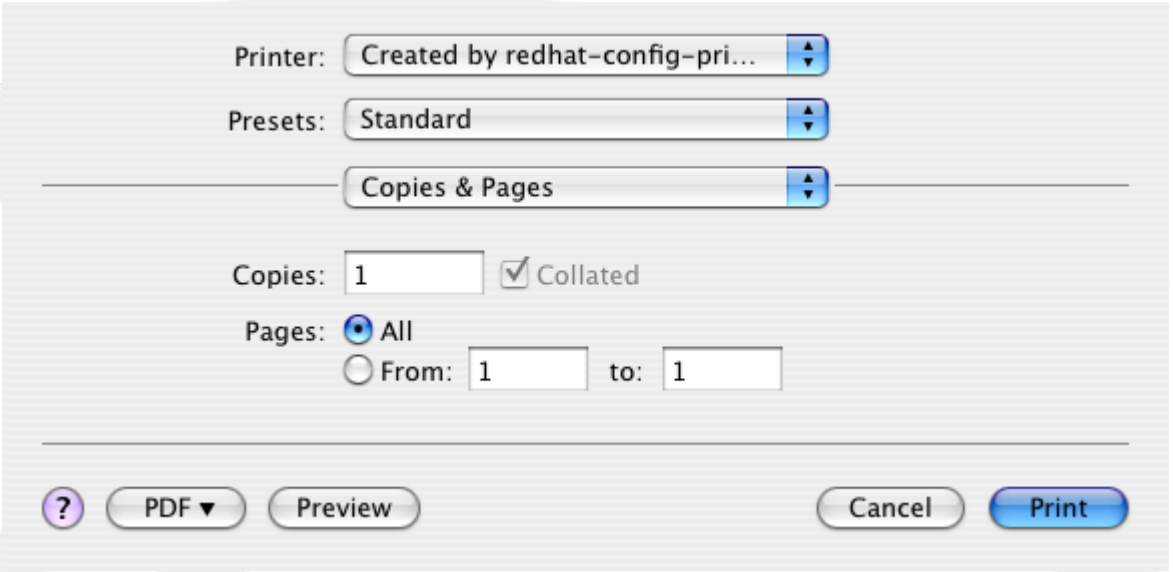




- Two dialogs: Page Setup and Print
- Custom tabs in print dialog
- Widget templates

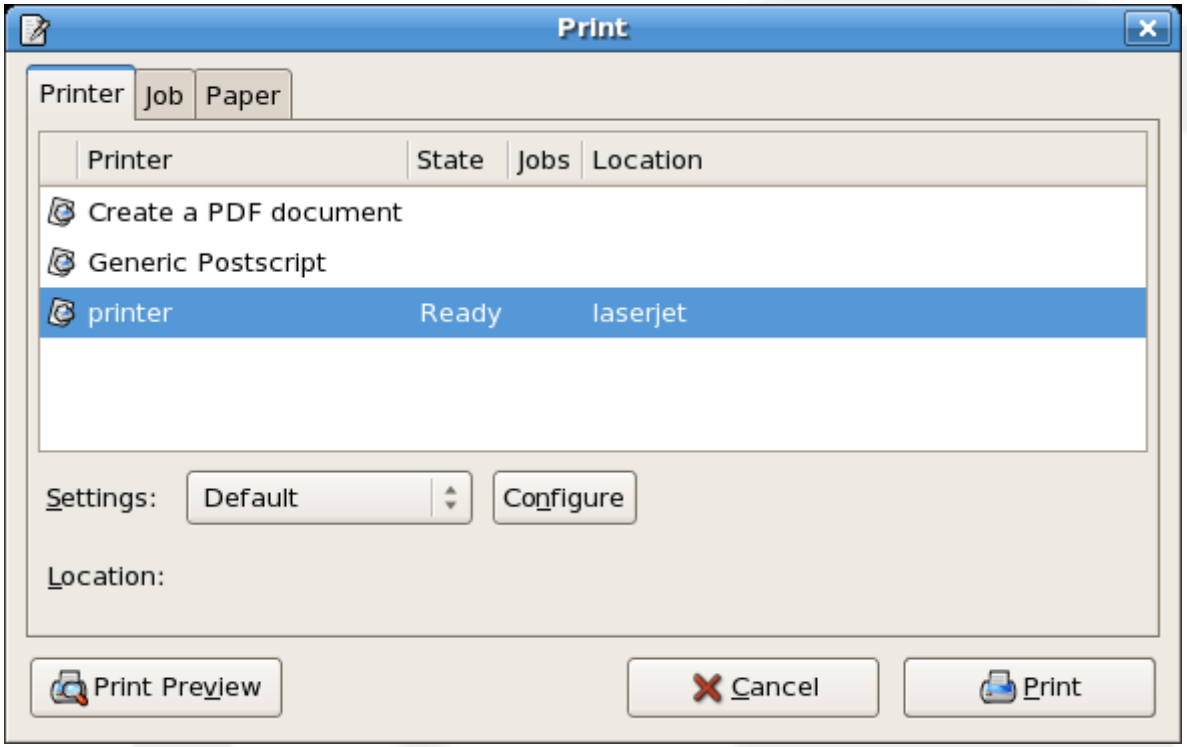
OSX Print dialogs





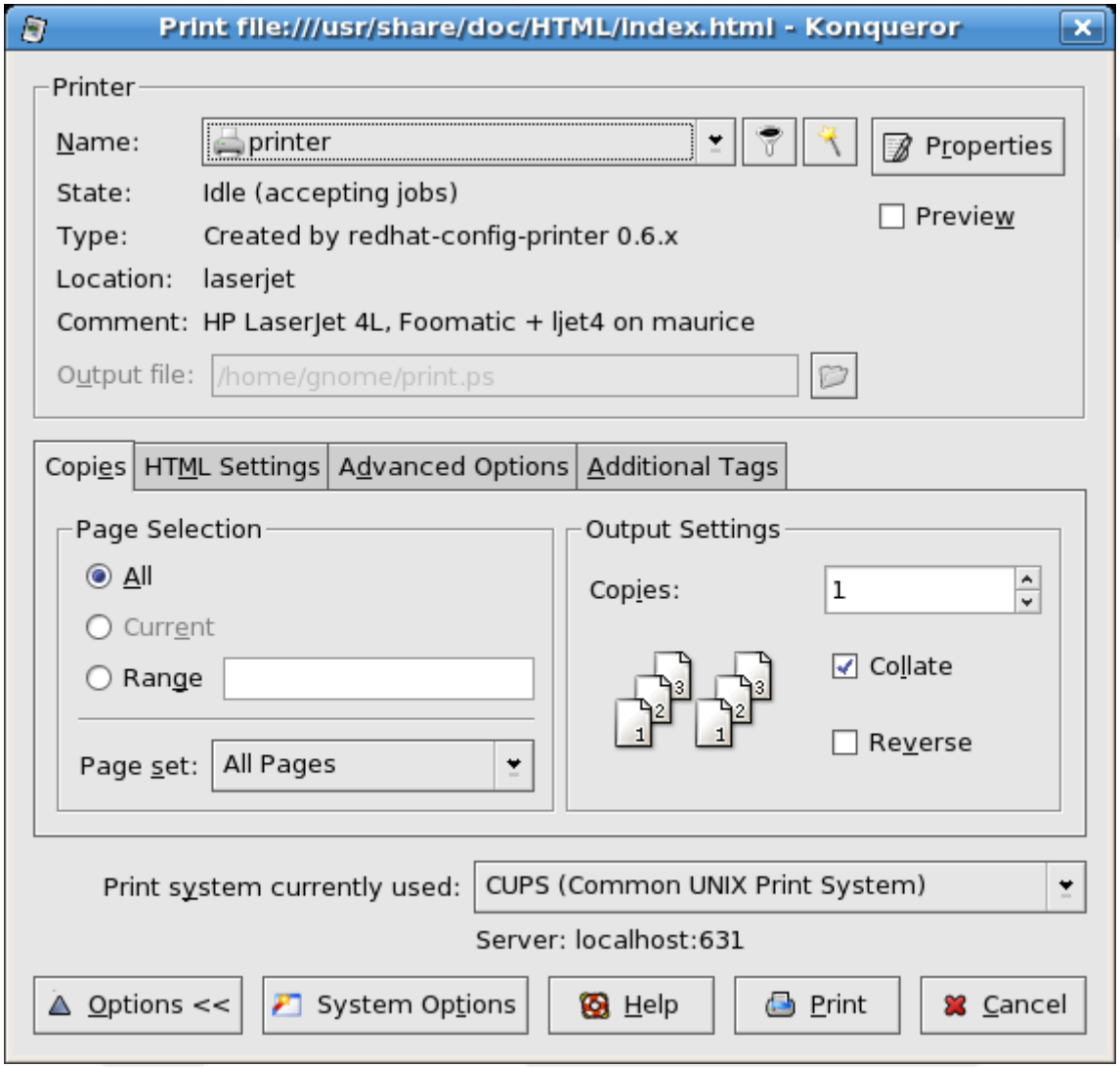
- Two dialogs
- Format for
- Custom tabs

GnomePrint print dialog



- No separate Page Setup dialog
- Limited printer settings

KDE print dialog



- No separate Page Setup Dialog
- Custom application tabs

Native printing

- Native dialogs
  - Printing looks the same as in other applications
  - Driver-specific UI additions work
- Native print system
  - Use underlying printer drivers
  - Common API with Cairo

Separate page setup dialog

- Used in both OSX and Windows
- Some applications do page layout before printing

Initial API proposal (libegg/print)

```
settings = egg_print_settings_new ();
g_object_set (settings,
              "output-filename", "test.pdf",
              NULL);

dialog = egg_print_dialog_new (settings);
egg_print_dialog_run (dialog);

job = egg_print_job_new (settings);
cr = egg_print_job_create_cairo (job);

cairo_rectangle (cr, 100, 100, 200, 200);
cairo_fill (cr);

cairo_show_page (cr);
cairo_destroy (cr);

egg_print_job_end (job);
```

New approach: GtkPrintOperation

- Each print operation allocates a print operation object
- Connect to signals (or inherit and override) to handle page drawing
- Automatically handles all the settings affecting the print loop

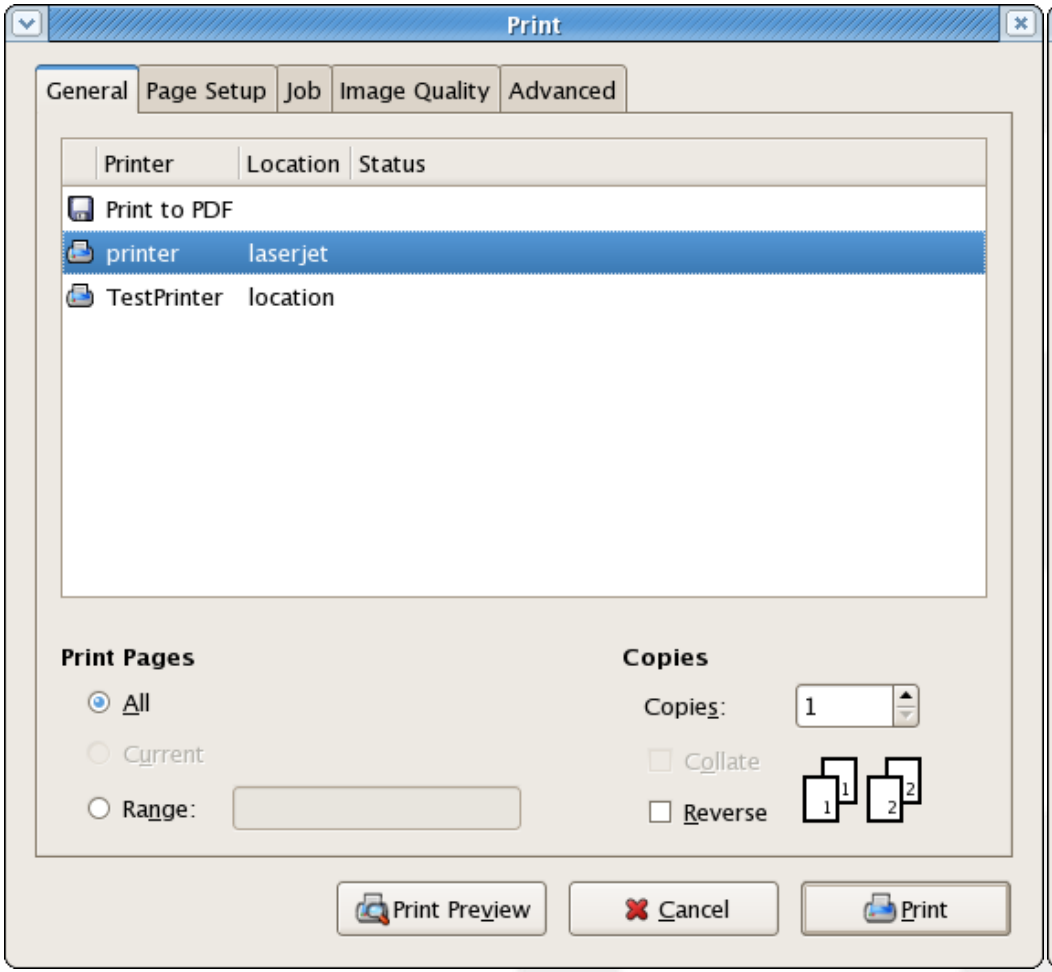
Simple example

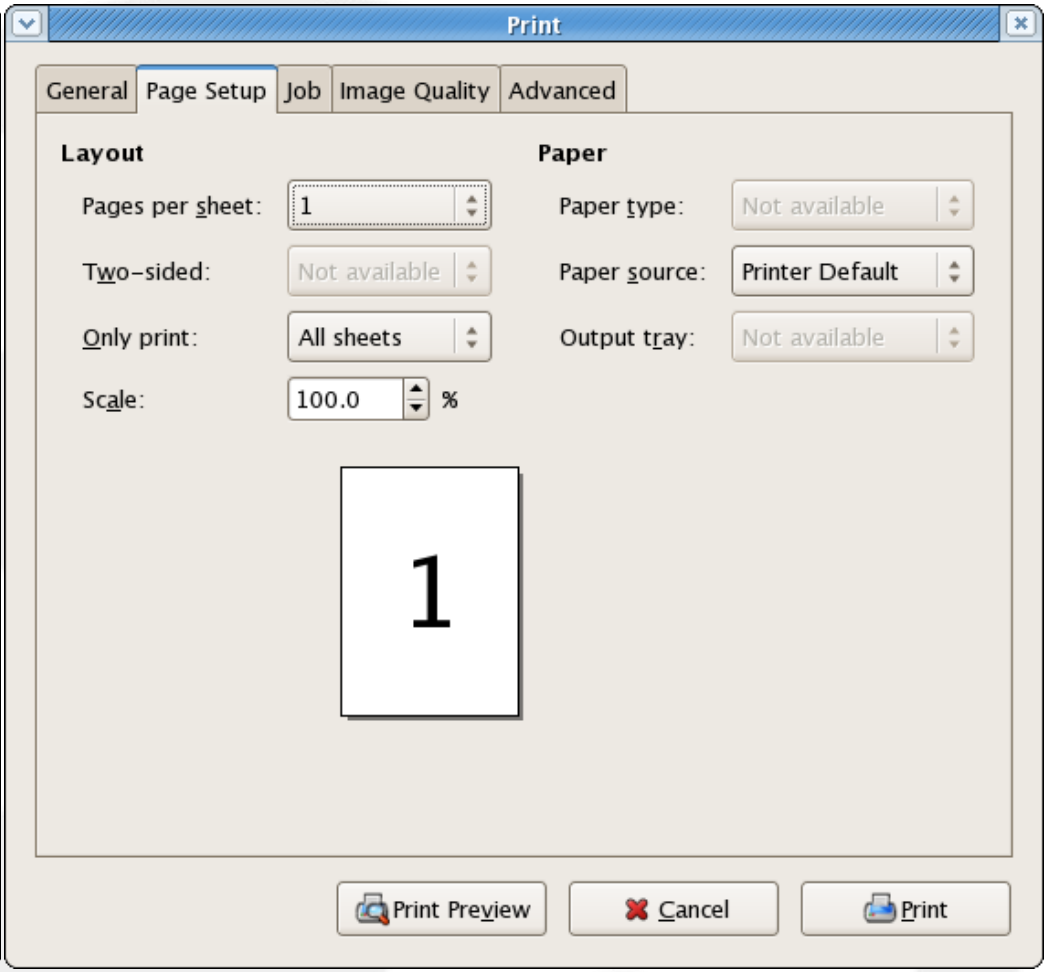
```
GtkPrintOperation *op;
GtkPrintOperationResult res;
GtkPrintSettings *settings;

op = gtk_print_operation_new ();

settings = get_last_print_settings ();
gtk_print_operation_set_print_settings (op, settings);

gtk_print_operation_set_n_pages (op, 1);

gtk_print_operation_set_unit (op, GTK_UNIT_MM);

g_signal_connect (op, "draw_page", G_CALLBACK (draw_page), NULL);

res = gtk_print_operation_run (op, GTK_PRINT_OPERATION_ACTION_PRINT_DIALOG,
                               NULL, NULL);
if (res == GTK_PRINT_OPERATION_RESULT_APPLY) {
  settings = gtk_print_operation_get_print_settings (op);
  save_print_settings (settings);
}
```

Simple Example (cont)

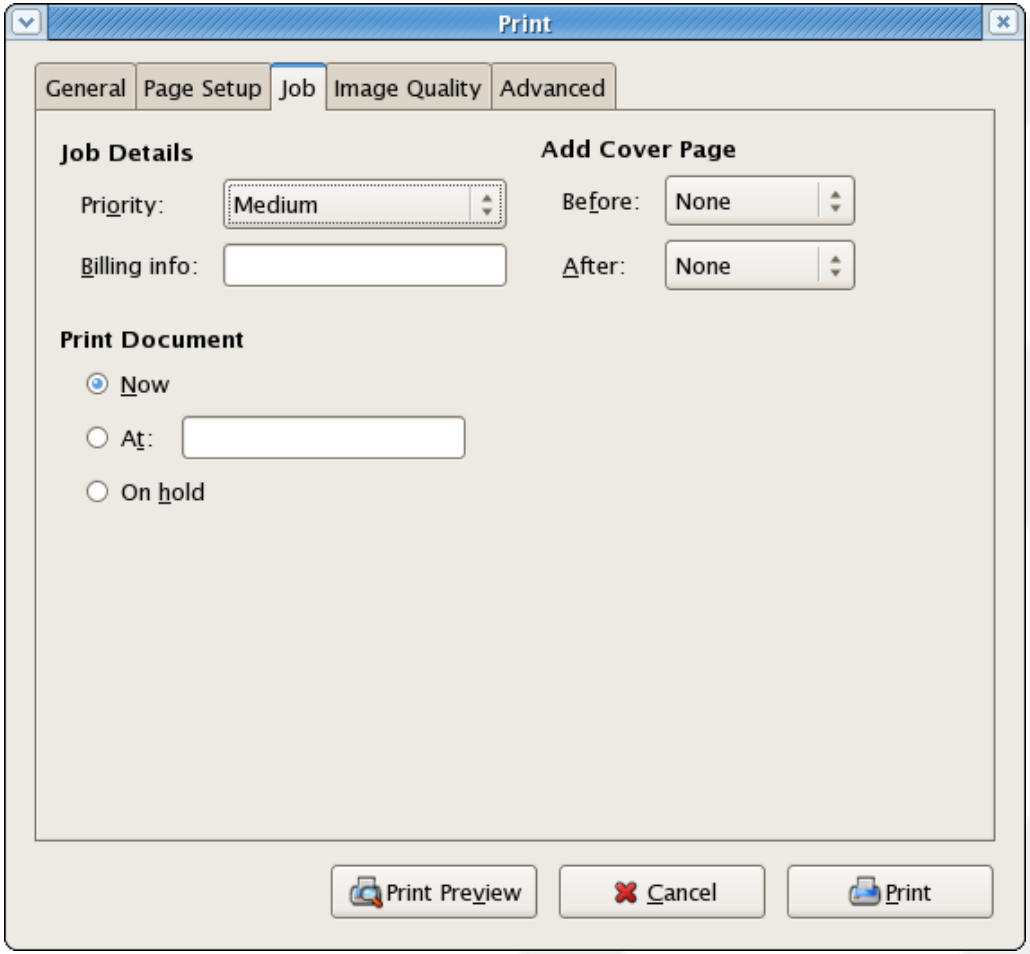
```
static void
draw_page (GtkPrintOperation *operation,
           GtkPrintContext *context,
           int page_nr)
{
  cairo_t *cr = gtk_print_context_get_cairo_context (context);

  /* Draw a red rectangle, as wide as the paper (inside the margins) */
  cairo_set_source_rgb (cr, 1.0, 0, 0);
  cairo_rectangle (cr, 0, 0, gtk_print_context_get_width (context), 50);
  cairo_fill (cr);

  /* Draw path */
  cairo_set_source_rgb (cr, 0, 0, 0);
  cairo_move_to (cr, 90, 75);
  cairo_line_to (cr, 60, 80);
  cairo_curve_to (cr, 40, 70, 65, 65, 70, 60);

  cairo_set_line_join (cr, CAIRO_LINE_JOIN_ROUND);
  cairo_set_line_width (cr, 5);

  cairo_stroke (cr);

}
```

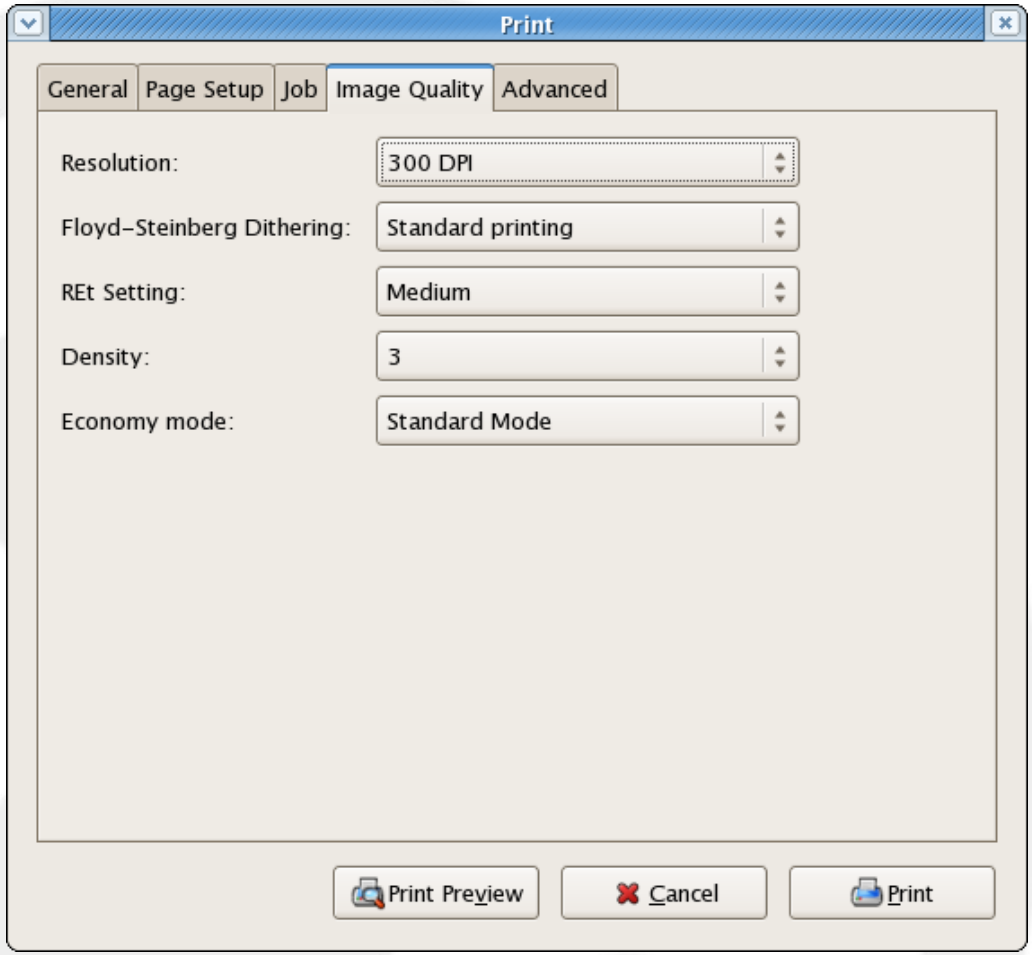
Simple Example (cont)





Simple Example (cont)





The Print Loop

- Emit **begin_print**
- Emit **paginate** (if connected) until it returns FALSE
- For each page that needs to be rendered
  - Emit **request_page_setup**
  - Emit **draw_page**
- Emit **end_print**
- Until print job finished, emit **status_changed** when job status changes
  - If *track-print-status* is TRUE this will also monitor the job status after spooling

Rendering text

- Text rendering is done using Pango
- gtk_widget_create_pango_layout() is only usable for screen targets
- Create layouts for printing using gtk_print_context_create_pango_layout()
- Don't reuse screen metrics, measure text in **begin_print** or **paginate**

```
layout = gtk_print_context_create_pango_layout (context);
pango_layout_set_text (layout, "Hello World! Printing is easy", -1);
desc = pango_font_description_from_string ("sans 28");
pango_layout_set_font_description (layout, desc);
pango_font_description_free (desc);
cairo_move_to (cr, 30, 20);
pango_cairo_show_layout (cr, layout);
```

Page Setup

- gtk_print_operation_set_default_page_setup() selects the default paper size, orientation and margins
- **request_page_setup** signal handler can modify this on a per-page basis
- **draw_page** cairo coordinate system:
  - Automatically rotated to the page orientation
  - normally inside the printer margins, but gtk_print_operation_set_use_full_page() changes
  - Unit in device pixels by default, but gtk_print_operation_set_unit() lets you select units

Async operations

- Some applications need to run a non-blocking print operation
- Use gtk_print_operation_set_allow_async()
- gtk_print_operation_run() can return GTK_PRINT_OPERATION_RESULT_IN_PROGRESS
- Connect to **done** signal for result/error handling
- Async not supported on all platforms (but **done** will still be emitted)

Printing results

- Printing is “done” when the print data is spooled
- The result will be returned from the run() function and in the arguments of the **done** signal
- GError available from gtk_print_operation_get_error()
- Possible results:
  - GTK_PRINT_OPERATION_RESULT_ERROR
  - GTK_PRINT_OPERATION_RESULT_APPLY
  - GTK_PRINT_OPERATION_RESULT_CANCEL
  - GTK_PRINT_OPERATION_RESULT_PREVIEW
  - GTK_PRINT_OPERATION_RESULT_IN_PROGRESS

Print Job Status

- You can track status updates with the **status_changed** signal and:
  - gtk_print_operation_get_status()
  - gtk_print_operation_get_status_string()
- Useful for updating a progress dialog or a status bar
- gtk_print_operation_is_finished() checks if the state is GTK_PRINT_STATUS_FINISHED or GTK_PRINT_STATUS_FINISHED_ABORTED
- Use gtk_print_operation_set_track_print_status() to get status tracking of the job in the printer queue.
- gtk_print_operation_set_show_progress()

Export to pdf

- Print to file is available in the print dialog, invisible to the application
- However, its sometimes useful to use the print code to generate a pdf file:

```
op = gtk_print_operation_new ();
// Set up op
gtk_print_operation_set_export_filename (print,
                                         "test.pdf");
res = gtk_print_operation_run (print,
                GTK_PRINT_OPERATION_ACTION_EXPORT,
                NULL, NULL);
```

Custom Widgets in Print Dialog

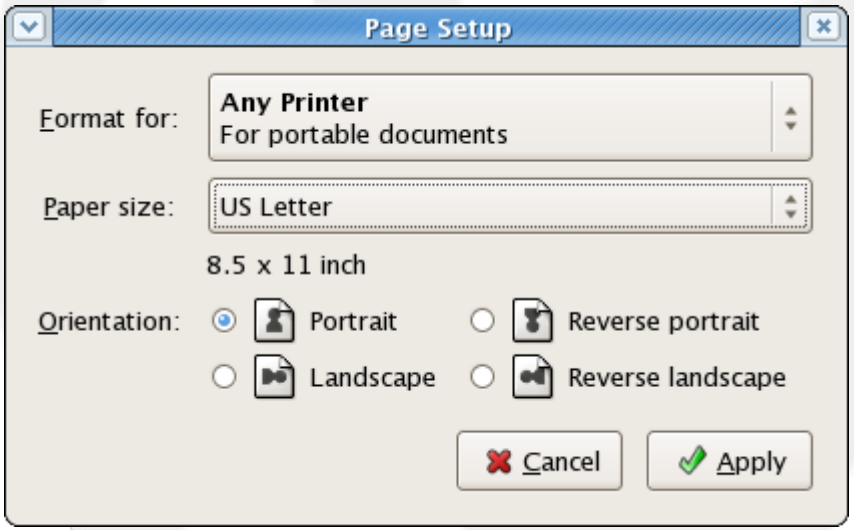
- You can adding a custom tab to the Print dialog:
  - Return a widget from the **create_custom_widget** signal handler
  - Save data from the widgets in the **custom_widget_apply** signal handler
- Even works in windows
- gtk_print_operation_set_custom_tab_label() allows you to set a non-default tab name

Preview

- The native Gtk dialog has a preview button
- Apps can manually start a preview by starting an operation with GTK_PRINT_OPERATION_ACTION_PREVIEW
- Default preview handler uses external viewer
- Its possible to override preview handling

Page Setup Dialog

- Call gtk_print_run_page_setup_dialog()
- Used to select:
  - Paper size
  - Orientation
  - Printer specific margins



Unix-only API

- GtkPrintOperation
  - High-level
  - Portable
  - Draw using cairo

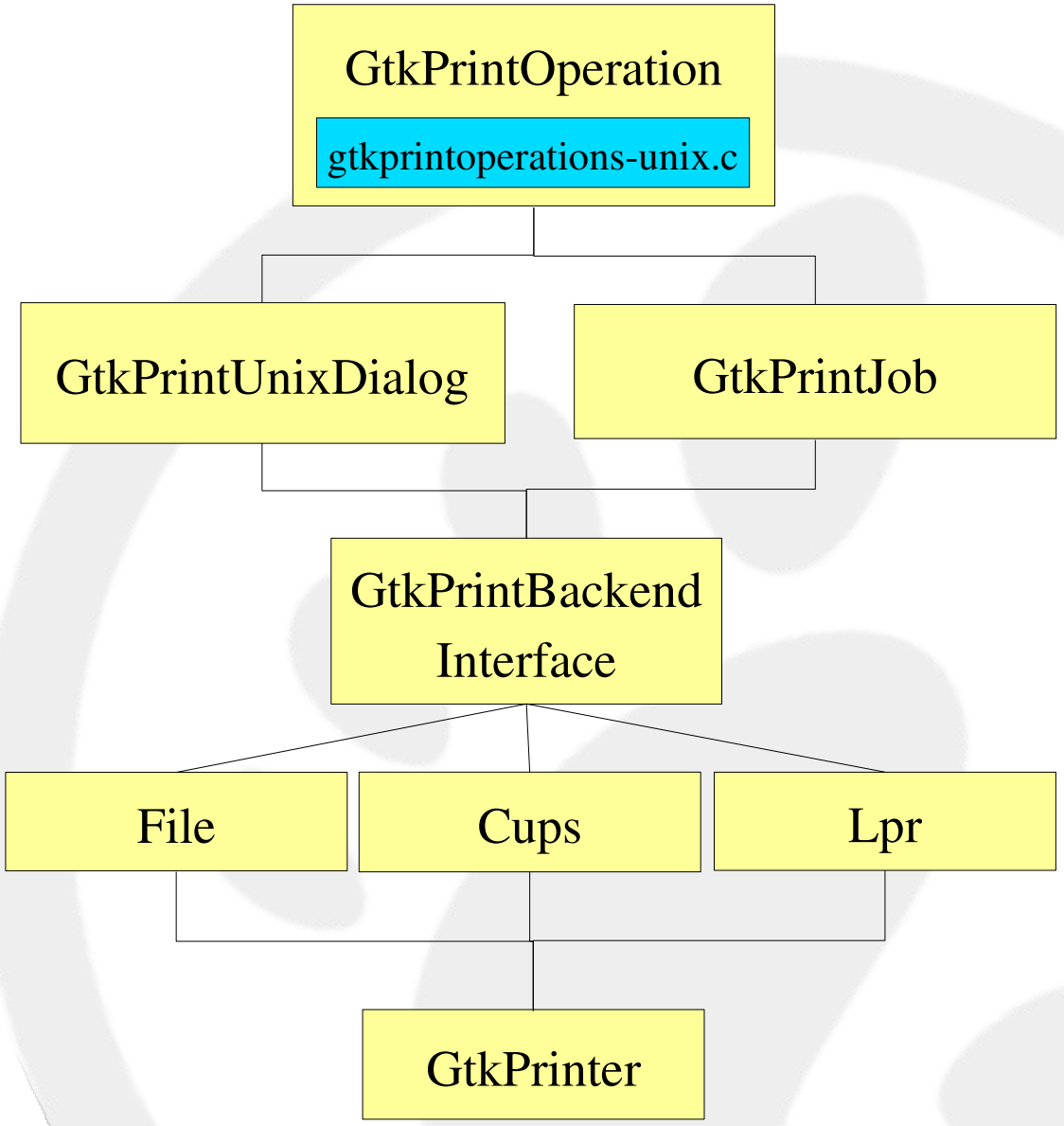
But... Applications like OOo wants to use the Gtk+ print dialog as a “native” print dialog

- Use module gtk+-unix-print-2.0.pc
- Only supported on unix-like OSes
- Feed app-generated postscript to the printer

Internals

- Platforms must define these symbols
  - `_gtk_print_operation_platform_backend_preview_end_page()`
  - `_gtk_print_operation_platform_backend_resize_preview_surface()`
  - `_gtk_print_operation_platform_backend_launch_preview()`
  - `_gtk_print_operation_platform_backend_preview_start_page()`
  - `_gtk_print_operation_platform_backend_create_preview_surface()`
  - `_gtk_print_operation_platform_backend_run_dialog`
- Optional Symbol
  - `_gtk_print_operation_platform_backend_run_dialog_async()` is not available on Win32
- These symbols are defined in gtkprintoperation-unix.c for Unix like platforms

Unix Dialog Internals



Unix Dialog Flow

- Dialog requests all backends modules
  - The gtk-print-backends setting defines which modules to load
- Dialog iterates each backend and requests a list of printers using `gtk_print_backend_get_printer_list()`
  - may return NULL (list can be generated ASYNC)
  - Signals printer_added, printer_removed, printer_list_changed, printer_list_is_done and printer_status_changed, keep the dialog up to date

Unix Dialog Flow (cont.)

- Selecting a printer
  - Dialog attaches to the details-acquired signal
  - printer_request_details() virtual method is called
  - Printer completes the GtkPrintOptions
  - Dialog uses options to set up the options tabs
- Printing
  - A GtkPrintJob is created from the printer and a cairo surface is created from the job
  - The backend streams the cairo commands to an output format (usually PS or PDF)

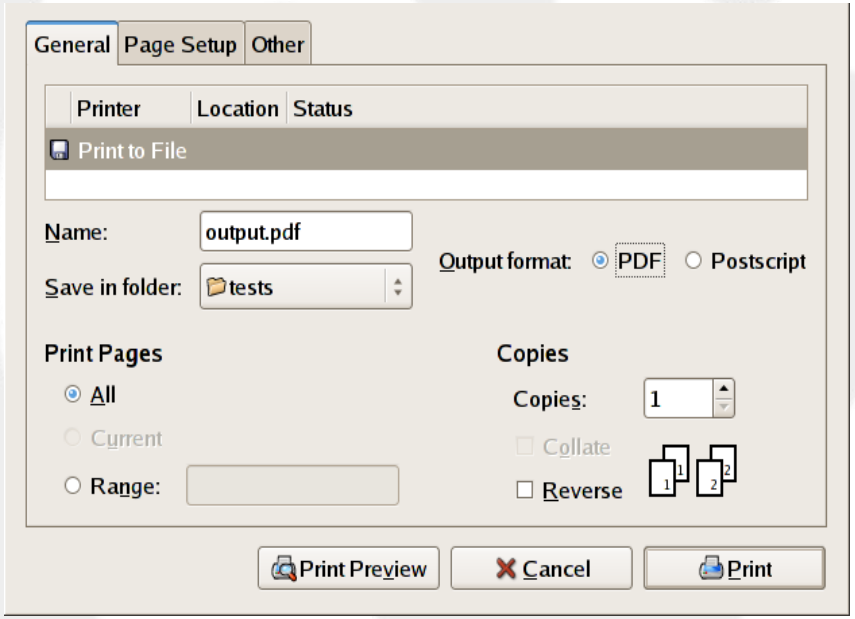
GtkPrintBackend Overview

- GtkPrinter and GtkPrintJob are mostly just interfaces into a GtkPrintBackend

```
/* Global backend methods: */
void                  (*request_printer_list)...
void                  (*print_stream)...

/* Printer methods: */
void                  (*printer_request_details)...
cairo_surface_t *     (*printer_create_cairo_surface)...
GtkPrinterOptionSet * (*printer_get_options)...
gboolean              (*printer_mark_conflicts)...
void                  (*printer_get_settings_from_options)...
void                  (*printer_prepare_for_print)...
GList  *              (*printer_list_papers)...
void                  (*printer_get_hard_margins)...
GtkPrintCapabilities  (*printer_get_capabilities)...

/* Signals */
void                  (*printer_list_changed)...
void                  (*printer_list_done)...
void                  (*printer_added)...
void                  (*printer_removed)...
void                  (*printer_status_changed)...
```

FILE Backend

- Simple Backend
- Prints to PDF or PS
- Creates a custom setting for the file name
  - A file chooser widget is added to the main tab



FILE Backend – Adding Printers

```
static void
gtk_print_backend_file_init (GtkPrintBackendFile *backend)
{
  GtkPrinter *printer;

  printer = g_object_new (GTK_TYPE_PRINTER,
              "name", _("Print to File"),
              "backend", backend,
              "is-virtual", TRUE,
              NULL);

  gtk_printer_set_has_details (printer, TRUE);
  gtk_printer_set_icon_name (printer, "gtk-floppy");
  gtk_printer_set_is_active (printer, TRUE);

  gtk_print_backend_add_printer (GTK_PRINT_BACKEND (backend), printer);
  g_object_unref (printer);

  gtk_print_backend_set_list_done (GTK_PRINT_BACKEND (backend));
}
```

FILE Backend – Creating a Surface

```
static cairo_surface_t *
file_printer_create_cairo_surface (GtkPrinter       *printer,
                                   GtkPrintSettings *settings,
                                   gdouble           width,
                                   gdouble           height,
                                   GIOChannel       *cache_io)
{
  cairo_surface_t *surface;
  OutputFormat format;

  format = format_from_settings (settings);

  if (format == FORMAT_PDF)
    surface = cairo_pdf_surface_create_for_stream (_cairo_write, cache_io, width, height);
  else
    surface = cairo_ps_surface_create_for_stream (_cairo_write, cache_io, width, height);

  /* TODO: DPI from settings object? */
  cairo_surface_set_fallback_resolution (surface, 300, 300);

  return surface;
}
```

FILE Backend – Caching cairo data

```
static cairo_status_t
_cairo_write (void                *closure,
              const unsigned char *data,
              unsigned int         length)
{
  GIOChannel *io = (GIOChannel *)closure;
  gsize written;
  GError *error;

  error = NULL;

  while (length > 0)
    {
      g_io_channel_write_chars (io, data, length, &written, &error);

      if (error != NULL)
        {
          g_error_free (error);
          return CAIRO_STATUS_WRITE_ERROR;
        }

      data += written;
      length -= written;
    }

  return CAIRO_STATUS_SUCCESS;
}
```

FILE Backend – Writting out the final product

```
static void
gtk_print_backend_file_print_stream (GtkPrintBackend
*print_backend,
                     GtkPrintJob            *job,
                     GIOChannel             *data_io,
                     GtkPrintJobCompleteFunc callback,
                     gpointer                user_data,
                     GDestroyNotify          dnotify)
{
  GError *internal_error = NULL;
  GtkPrinter *printer;
  _PrintStreamData *ps;
  GtkPrintSettings *settings;
  gchar *filename = NULL;

  printer = gtk_print_job_get_printer (job);
  settings = gtk_print_job_get_settings (job);

  ps = g_new0 (_PrintStreamData, 1);
  ps->callback = callback;
  ps->user_data = user_data;
  ps->dnotify = dnotify;
  ps->job = g_object_ref (job);
  ps->backend = print_backend;
```

FILE Backend – Writting out the final product (cont.)

```
  internal_error = NULL;
  filename = filename_from_settings (settings);

  ps->target_io = g_io_channel_new_file (filename, "w", &internal_error);

  g_free (filename);

  if (internal_error == NULL)
    g_io_channel_set_encoding (ps->target_io, NULL, &internal_error);

  if (internal_error != NULL)
    {
      file_print_cb (GTK_PRINT_BACKEND_FILE (print_backend),
                     internal_error, ps);

      g_error_free (internal_error);
      return;
    }

  g_io_add_watch (data_io,
                  G_IO_IN | G_IO_PRI | G_IO_ERR | G_IO_HUP,
                  (GIOFunc) file_write,
                  ps);
}
```

CUPS Backend

- A lot more complicated than the FILE backend
- CUPS Convenience API is not ASYNC
  - gnome-print solved this by using threads
  - gtk-print solves this without requiring threading
    - CUPS convenience API was replicated as a state machine (gtkcupsutils.c)
    - Backend calls this state machine from a source

CUPS Backend – Adding Printers

```
static gboolean
cups_request_printer_list (GtkPrintBackendCups *cups_backend)
{
  GtkCupsRequest *request;
  static const char * const pattrs[] = /* Attributes we're interested in */
    {
      "printer-name",
      "printer-uri-supported",
      "member-uris"
    };

  cups_backend->list_printers_pending = TRUE;
  request = gtk_cups_request_new (NULL,
                                  GTK_CUPS_POST,
                                  CUPS_GET_PRINTERS,
                                  NULL, NULL, NULL);

  gtk_cups_request_ipp_add_strings (request, IPP_TAG_OPERATION, IPP_TAG_KEYWORD,
                                    "requested-attributes", G_N_ELEMENTS (pattrs),
                                    NULL, pattrs);

  cups_request_execute (cups_backend, request,
                        (GtkPrintCupsResponseCallbackFunc) cups_request_printer_list_cb,
                        request, NULL);

  return TRUE;
}
```

CUPS Backend – Adding Printer (cont.)

```
static void
cups_request_printer_list_cb (GtkPrintBackendCups *cups_backend,
                              GtkCupsResult       *result,
                              gpointer             user_data)
{
  removed_printer_checklist = gtk_print_backend_get_printer_list (backend);

  response = gtk_cups_result_get_response (result);

  for (attr = response->attrs; attr != NULL; attr = attr->next)
    {
      while (attr != NULL && attr->group_tag != IPP_TAG_PRINTER)
        attr = attr->next;
        {
          /* create printers */
           gtk_print_backend_add_printer (backend, printer);
          if (gtk_printer_is_new (printer))
            g_signal_emit_by_name (backend, "printer-added", printer);
        }
    }

  /*code to figure out which printers were removed */
  if (list_has_changed)
    g_signal_emit_by_name (backend, "printer-list-changed");
```

CUPS Backend – Writing out the final product

```
static void
gtk_print_backend_cups_print_stream (GtkPrintBackend         *print_backend,
                                     GtkPrintJob             *job,
                                     GIOChannel              *data_io,
                                     GtkPrintJobCompleteFunc  callback,
                                     gpointer                 user_data,
                                     GDestroyNotify           dnotify)
{
  GtkPrinterCups *cups_printer;
  CupsPrintStreamData *ps;
  GtkCupsRequest *request;
  GtkPrintSettings *settings;
  const gchar *title;

  cups_printer = GTK_PRINTER_CUPS (gtk_print_job_get_printer (job));
  settings = gtk_print_job_get_settings (job);

  request = gtk_cups_request_new (NULL,
                                  GTK_CUPS_POST,
                                  IPP_PRINT_JOB,
                                  data_io,
                                  NULL,
                                  cups_printer->device_uri);
```

CUPS Backend – Writing out the final product (cont.)

```
  gtk_cups_request_ipp_add_string (request, IPP_TAG_OPERATION, IPP_TAG_URI,
                                   "printer-uri",NULL, cups_printer->printer_uri);

  gtk_cups_request_ipp_add_string (request, IPP_TAG_OPERATION, IPP_TAG_NAME,
                                   "requesting-user-name", NULL, cupsUser());

  title = gtk_print_job_get_title (job);
  if (title)
    gtk_cups_request_ipp_add_string (request, IPP_TAG_OPERATION, IPP_TAG_NAME,
                                     "job-name", NULL, title);

  gtk_print_settings_foreach (settings, add_cups_options, request);

  ps = g_new0 (CupsPrintStreamData, 1);
  ps->callback = callback;
  ps->user_data = user_data;
  ps->dnotify = dnotify;
  ps->job = g_object_ref (job);

  cups_request_execute (GTK_PRINT_BACKEND_CUPS (print_backend),
                        request,
                        (GtkPrintCupsResponseCallbackFunc) cups_print_cb,
                        ps,
                        (GDestroyNotify)cups_free_print_stream_data);
}
```