الجامعة الافتراضية السورية
SYRIAN VIRTUAL UNIVERSITY

النظم الموزعة

الدكتور إياء عويشق

Books

النظم الموزعة

الدكتور إباء عويشق

من منشورات الجامعة الافتراضية السورية

الجمهورية العربية السورية 2018

Distributed Systems

Ibaa Oueishek

Publications of the Syrian Virtual University (SVU)

Syrian Arab Republic, 2018

الفهرس

الفصل الأول

لمحة عامة عن النظم الموزعة

سنقوم في هذا الفصل بتقديم التعريف الأساسي والأكثر استخداماً للنظمة الموزعة، بالإضافة إلى عرض الخصائص الأساسية للأنظمة الموزعة وعدد من الأمثلة عن هذه الأنظمة بالإضافة إلى التحديات الرئيسية التي تواجه تصميم وتحقيق هذه الأنظمة ووضعها في الاستخدام الفعلي.

تعريف أساسي أولي:

النظام الموزع هو نظام تتوضع فيه المكونات على حواسيب متعددة متصلة عبر شبكة، وتتصل هذه المكونات وتنسق أعمالها المختلفة عن طريق تبادل الرسائل.

ملاحظة هامة: هذا التعريف أولي وتنقصه الدقة، وسنعطي تعريفاً أكثر دقة في الفصل الثاني بعد فحص الأوجه المختلفة للأنظمة الموزعة.

1. الخصائص الأساسية للأنظمة الموزعة

يقود هذا التعريف مباشرة إلى الخصائص الرئيسية المميزة للأنظمة الموزعة: توارد المكونات أوعملها في نفس الوقت، وعدم وجود ميقاتية موحدة، واستقلال أعطال المكونات المختلفة.

التوارد concurrency

وهي خاصة ناتجة بشكل طبيعي عن وجود عدة حواسيب ضمن النظام الموزع، وبالتالي يمكن لكل من هذه الحواسيب والمستخدمين الذين يعملون عليها القيام بأعمال متواردة. لا بل يجب أن يكون العمل متوارداً وإلا لما كانت هناك أية جدوى من بناء هذه الأنظمة. وسنجد هنا نفس المشكلات التي كانت تظهر في الأنظمة المركزية والحاجة للتنسيق والتزامن بين الإجرائيات المتواردة، مع ملاحظة أن العديد من الآليات التي كانت تستخدم في تلك الأنظمة غير مقبولة هنا ولا يمكن استخدامها نظراً لكونها تعتمد على وجود ذاكرة مشتركة (كالسيمافور مثلاً) وهي غير متوافرة في الأنظمة الموزعة.

عدم وجود ميقاتية موحدة

تحتاج البرامج والإجرائيات إلى تنسيق عملها عبر تبادل الرسائل، ويحتاج التنسيق على معرفة اللحظة الزمنية التي تقع فيها الأحداث. ولكن لا يمكن للحواسيب المتصلة عبر الشبكة أن تزامن ميقاتياتها فعلياً، وبالتالي لا يوجد هناك اتفاق بين الإجرائيات المختلفة على لحظة زمنية صحيحة وشاملة عبر كل الشبكة. وهذه نتيجة مباشرة لكون الطريقة الوحيدة للاتصال هي إرسال الرسائل عبر الشبكة.

الفشل المستقل للمكونات المختلفة

يمكن لكافة المنظومات الحاسوبية أن تفشل وتحصل فيها أخطاء بأشكال مختلفة. إذ تتسبب أخطاء الشبكة في عزل الحواسيب المتصلة بها، ولكن دون ان تؤدي إلى توقف هذه الحواسيب عن العمل. ومن الصعب على البرامج التي تعمل أن تميز بين حالات وجود عطل في الشبكة أو البطئ الشديد. وبنفس الطريقة، فإن انتشار المعلومة عن حصول عطل في حاسب، أو التوقف غير المتوقع لأحد البرامج يحتاج إلى وقت غير محدد. ويمكن للأخطاء أن تحصل بشكل مستقل وبدون أي ترابط بين بعضها البعض، وهذه إحدى أهم خصائص الأنظمة الموزعة.

يبذل مصممو الأنظمة الموزعة قصارى جهدهم لتقديم خاصة الشفافية للتطبيقات والمستخدمين، ويقصد بذلك أن أغلب الاتصالات التي تتم بين مكونات النظام هي غير مرئية بالنسبة للمستخدم أو للتطبيق. ومثال على ذلك مخدمات أنظمة الملفات، إذ يظهر نظام التشغيل للمستخدم أو للتطبيق أن نظام الملفات المتوضع غلى المخدم هو مطابق تماماً من الناحية الوظيفية للنظام المحلي. وبالتالي لا يشعر المستخدم البتة بوجود عمليات النفاذ هذه عبر الشبكة.

ويعتبر مفهوم الشفافية أحد أهم المفاهيم والتحديات في النظمة الموزعة، ويمكن تصنيف الشفافية ضمن عدة أوجه:

- **شفافية النفاذ access transparency**: ويقصد بها إمكانية الوصول إلى الموارد المحلية والبعيدة باستخدام تعليمات متماثلة.
- **شفافية الموقع location transparency**: ويقصد ذلك إمكانية النفاذ إلى الموارد بدون معرفة توضعها الجغرافي أو موقعها على الشبكة (على سبيل المثال عنوان المبنى أو عنوان الشبكة).
- **شفافية التوارد concurrency transparency**: ونعني بذلك السماح لعدة إجرائيات بالعمل بشكل متوارد واستخدام الموارد المشتركة بدون وجود تضارب بينها.
- **شفافية النسخ المتعدد replication transparency**: ويقصد بها تمكين استخدام نسخ متعددة من نفس الموارد بهدف زيادة الوثوقية أو تحسين الإدارء وبدون أن يظهر ذلك للمستخدمين أو للتطبيقات.
- **شفافية الأعطال Failure transparency**: وهي تسمح بإخفاء الأعطال بحيث يمكن للمستخدمين والإجرائيات الاستمرار في أداء عملها رغم حصول توقف في مكونات برمجية أو عتادية.
- **شفافية التحرك mobility transparency**: وهي تسمح بحركة الموارد والمستخدمين ضمن النظام بدون التأثير على عمل المستخدمين أو التطبيقات.
- **شفافية الأداء performance transparency**: تسمح بإعادة تشكيل النظام بهدف تحسين الأداء استجابة لحصول تغيرات في الضغط والحمل على التجهيزات.
- **شفافية التوسع scaling transparency**: تسمح بتوسيع النظام والتطبيقاتعرضياً بدون حصول تغيير بنية النظام والتطبيقات.

2. تطور الأنظمة الموزعة

المحفزات التقانية لبناء الأنظمة الموزعة

شهدت سنوات التسعينيات من القرن العشرين نمواً متسارعاً في تطوير الأنظمة الموزعة والاعتماد عليها في العديد من التطبيقات. وقد كانت هناك العديد من المحفزات التكنولوجية التي غذت هذا التوجه ودفعته قدماً نحو الأمام وأهمها:

- التطور التقاني في مجال الإلكترونيات الصغرية، والذي أدى إلى ظهور تجهيزات صغيرة الحجم، وتتمع بقدرة معالجة وسعات تخزينية ممتازة. حتى أصبح الحاسوب المحمول في يومنا هذا يتمتع بإمكانات تفوق أكبر الحواسيب التي كانت مستخدمة في بداية التسعينيات.
- التطور التقاني في مجال الشبكات، ولا سيما الشبكات المحلية التي باتت قادرة على نقل المعطيات بسرعات هائلة تبلغ **10Gbps**. أي أن سرعة نقل المعطيات عبر الشبكة لم تعد أدنى بكثير من سرعة نقل المعطيات داخل الناقل الداخلي للحاسب. وحتى الشبكات بعيدة المدى أصبحت تعمل بسرعات عالية (يوجد الآن معيار دولي لسرعة **40 Gbps**، بالإضافة إلى شبكات قيد المعيرة تعمل بسرعة **100Gbps**)، وذلك بفضل الاعتماد على الألياف الضوئية التي تتميز بارتفاع الوثوقية والأداء والانخفاض المستمر في كلفتها.

الدوافع الأساسية لبناء الأنظمة الموزعة

هناك دوافع عدة لبناء الأنظمة الموزعة واستخدامها، ندرج فيما يلي أهمها:

وجود العديد من التطبيقات الموزعة بطبيعتها: فقد ظهرت الأنظمة الموزعة كانعكاس طبيعي للتوزع الموجود في المجتمع، إذ يتوزع الأشخاص والمعلومات جغرافياً. ومن المفترض أن تتلاءم التطبيقات والأنظمة مع هذا التوزع ليكون التطبيق أقرب من المستخدم وليس بالعكس فلا نحتاج إلى فرض مكان معين على المستخدم. وربما كانت التطبيقات المحمولة التي بدأت بالظهور مع انتشار الهواتف النقالة أفضل مثال على التوجه نحو توزيع الحوسبة وتوزيع المعلومات ومصادرها على نطاق واسع. ويترافق هذا التوزع مع الحاجة في العديد من الحالات إلى تقديم نظرة شاملة للتطبيق أو النظام، بالإضافة إلى دعم العديد من العمليات الشاملة أي التي يتدخل فيها العديد من المكونات المتباعدة للنظام.

نسبة الأداء إلى الكلفة: بسبب دعمها للتوارد، فإن الأنظمة الموزعة تقلل من نقاط الاختناق وتقدم أداء إجمالياً أفضل من الأنظمة المركزية. وبالتالي فإن نسبة الأداء إلى الكلفة تكون أفضل، سيما أن جل المكونات المستخدمة هي مكونات معيارية ومنخفضة الكلفة. ويجب أن ننتبه هنا إلى أن رفع نسبة الأداء إلى الكلفة لا يعني بالضرورة بناء حواسيب فائقة السرعة كما يمكن أن يخطر على بال البعض. فبناء الحواسيب الفائقة السرعة هو من اختصاص الأنظمة التفرعية **parallel systems**، ويحتاج عادة إلى شبكات وحواسيب وأنظمة تشغيل خاصة. وفي كثير من الأحيان لايكون بناء هذه الأنظمة اقتصادياً، وإنما حاجة لا بد منها لتحقيق عدد من المتطلبات الخاصة، كاستخدامها في الحوسبة فائقة الأداء.

التشارك في الموارد: فبإمكان الأنظمة الموزعة أن تدعم وبفاعلية عالية التشارك في المعلومات والموارد البرمجية والعتادية. والمقصود بمصطلح الموارد هنا هو المفهوم العام لكل ما يمكن لنظام التشغيل توفيره للتطبيقات، سواء أكانت عتادية كالطابعات أو زمن المعالجة أو وسائط التخزين، أو برمجية كالمكتبات البرمجية المتخصصة أو الملفات الخاصة بالمستخدمين أو محتويات قواعد البيانات. إذ تتيح هذه الأنظمة للمستخدمين الموجودين في مواقع مختلفة استخدام الموارد أينما كانت ضمن النظام.

المرونة وقابلية التوسع: يمكن بسهولة توسيع الأنظمة الموزعة عن طريقة إضافة مكونات جديدة. ولهذه الأنظمة ميزة إضافية كبيرة هي إمكانية تعديل أو توسيع النظام القائم بهدف التأقلم مع بيئة متغير وبدون التأثير على عمله. ويمكن لهذا التوسع أن يكون كمياً أي إضافة مكونات شبيهة لمكونات موجودة أو نوعياً أي إضافة مكونات غير موجودة مسبقاً.

الجاهزية ومقاومة الأعطال: إذ توجد لدى الأنظمة الموزعة إمكانية متابعة العمل رغم وجود أعطال ضمن النظام وذلك بسبب تعدد وحدات التخزين ووحدات المعالجة.

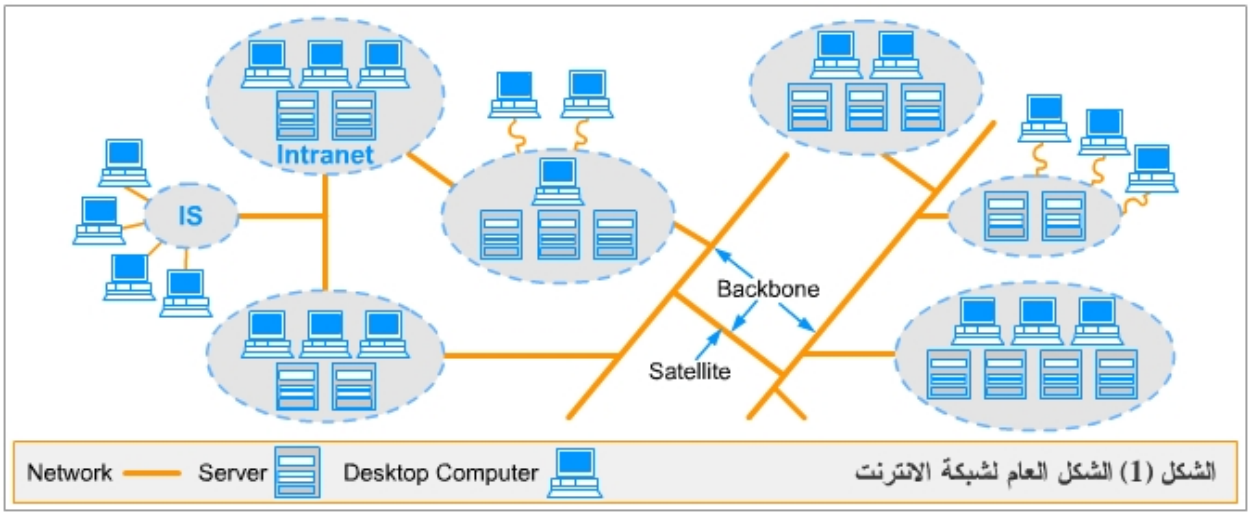
3. أمثلة على الأنظمة الموزعة

نورد فيما يلي ثلاثة أمثلة مباشرة للأنظمة الموزعة:

- الإنترنت.
- الإنترانت، وهي شبكة خاصة بمنظمة أو جهة ما وتستخدم في عملها بروتوكولات الإنترنت نفسها.
- الحوسبة المحمولة.

الإنترنت

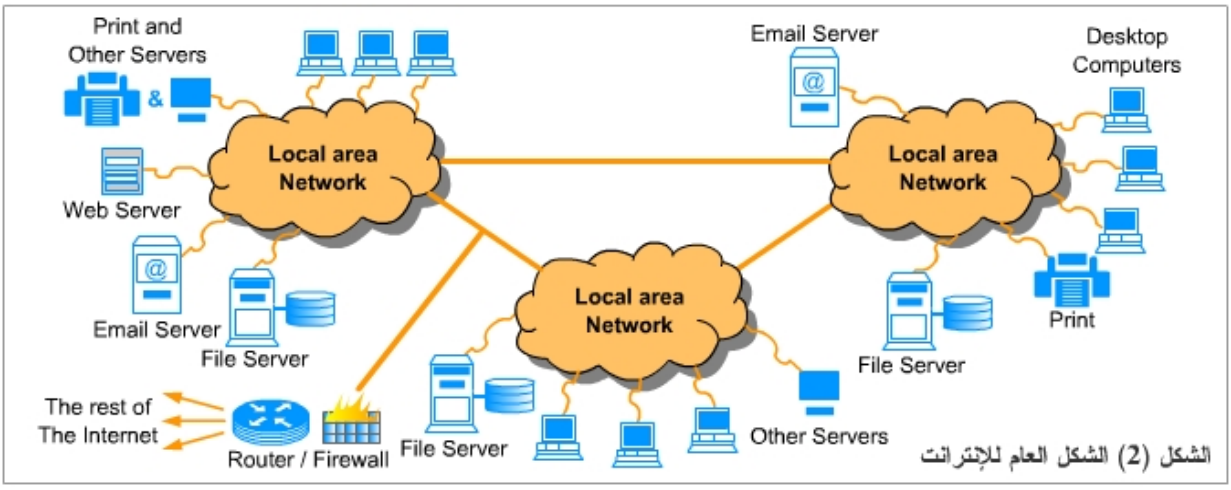
وهي أكبر نظام موزع موجود في العالم، وتعرف أيضاً باسم شبكة الشبكات، إذ تتكون في أساسها من عدد هائل من الشبكات التي ترتبط ببعضها البعض باستخدام بروتوكولات الإنترنت. وتعتبر مجموعة البروتوكولات الملحقة بالإنترنت والخاصة بتبادل المعلومات إنجازاً فنياً كبيراً، نظراً لكونها تسمح للتطبيقات بالتخاطب بين بعضها البعض بغض النظر عن مكان توضعها على الشبكة. ويظهر الشكل 1-1 البنيان العام لشبكة الإنترنت.



ومن أهم ميزات الإنترنت هو اتساع نطاقها، إذ يمكن للمستخدمين أينما كانوا في العالم استخدام الخدمات التقليدية المتاحة على الشبكة كخدمة الوب والبريد الإلكتروني ونقل الملفات، وهذه المجموعة من الخدمات غير منتهية نظرياً إذ يمكن إضافة خدمات جديدة في أية لحظة وجعلها متاحة عبر الشبكة. وقد سمح تطور أداء وسائط النقل والزيادة الكبيرة في عرض الحزمة بتقديم خدمات نقل الوسائط المتعددة كالبث الصوتي والمرئي عبر الشبكة. ولكن تبقى إمكانية دعم هذا النوع من التطبيقات محدودة بسبب متطلباتها الخاصة والتي لم تؤخذ في الاعتبار عند تصميم الشبكة.

الإنترانت

تعرف الإنترانت على أنها جزء من شبكة الإنترنت منفصل إدارياً ولديه حدود يمكن تشكيلها بحيث يتم تحديد سياسات أمنية محلية (هناك تعريف آخر أقل أكاديمية: الإنترانت هي شبكة داخلية خاصة تستخدم في توصيلها وعملها بروتوكولات الإنترنت). ويظهر الشكل 2 شبكة إنترانت نموذجية مكونة من عدة شبكات محلية مرتبطة ببعضها البعض باستخدام وصلات فقارية. وتقع مسؤولية تشكيل شبكة الإنترانت على عاتق المنظمة التي تقوم بإدارتها ويتراوح هذا التشكيل بين شبكة واحدة على موقع واحد إلى مجموعة مترابطة من الشبكات المحلية تنتمي إلى فروع للمنظمة منتشرة في عدة بلدان.



الشكل (2) الشكل العام للإنترانت

ترتبط الإنترانت عادة بشبكة الإنترنت عبر جهاز يعرف باسم الموجه **router**، مما يسمح للمستخدمين داخل الإنترنات باستخدام خدمات الإنترنت أينما كانت كالوب والبريد الإلكتروني. تسمح أيضاً لمشتركي للمستخدمين في الشبكات الأخرى بالوصول إلى الخدمات التي تقدمها. وتحتاج العديد من المنظمات إلى حماية خدماتها من من النفاذ غير المشروع للمستخدمين المسيئين، فعلى سبيل المثال لا يريد مشفى يصل شبكة الإنترانت الخاصة به مع الإنترنت أن يكون ذلك سبباً للوصول إلى البيانات الخاصة بالمرضى والحصول على نسخة منها. أو أن يؤدي ذلك إلى دخول برامج مؤذية إلى داخل الإنترانت والتأثير على عملها وإيقافها.

تستخدم شبكات الإنترانت في سبيل عزلها عن الإنترنت جهازاً يعرف باسم الجدار الواقي أو مانع النار **firewall** (يترجم أيضاً جدار ناري، ولكن هذه الترجمة توحي بأن المقصود هو جدار من النار، وفي الحقيقة هذا المصطلح لا يعني ذلك البتة. فالمصطلح **firewall** يستخدم لتسمية الحاجز الفاصل بين غرفة المحرك وغرفة السائق في السيارة بحيث لا تصل أية نار قد تشتعل في المحرك إلى غرفة السائق. وبالتالي فإن مصطلحات الجدار الواقي أو مانع النار هي أكثر صحة). ويستخدم الجدار الواقي في منع الرسائل غير المسموح بها من الدخول أو الخروج إلى الإنترنت، وهو يعتمد في ذلك على فحص الرسائل الداخلة والخارجة (مثلاً: عدم السماح بدخول الرسائل التي لا يكون مصدرها أحد فروع الإنترانت).

ويطرح استخدام وتشغيل الإنترانت العديد من النقاط التي ستناقش عبر هذا المقرر:

- خدمات الملفات التي تستخدم بغرض السماح للمستخدمين بالتشارك في البيانات.
- يمكن للجدار الواقي ألا يميز بين محاولات الدخول المسيئة والمحاولات المشروعة. وهو لا يكفي لوحده في تأمين الحماية اللازمة عندما يكون هناك تشارك في الموارد بين المستخدمين الداخليين والخارجيين. وهناك حاجة لاستخدام آليات أمن أكثر دقة سنراها لاحقاً.
- باعتبار أن الإنترانت هي ذات طابع مؤسساتي، وبالتالي فإن مواضيع كلفة استخدام الشبكة وبالذات تثبيت ودعم البرمجيات يلعب دوراً هاماً في إدارتها. وبالتالي هناك العديد من التصاميم التي ظهرت خلال السنوات الماضية تهدف إلى تخفيض هذه الأكلاف، مثل الزبائن النحيفة **thin clients**.

الحوسبة المحمولة

أدت التطورات المتلاحقة في التقانات الصغرية والشبكات اللاسلكية إلى حصول زيادة كبيرة في الإمكانات الحاسوبية المتاحة ضمن التجهيزات النقالة، ومنها:

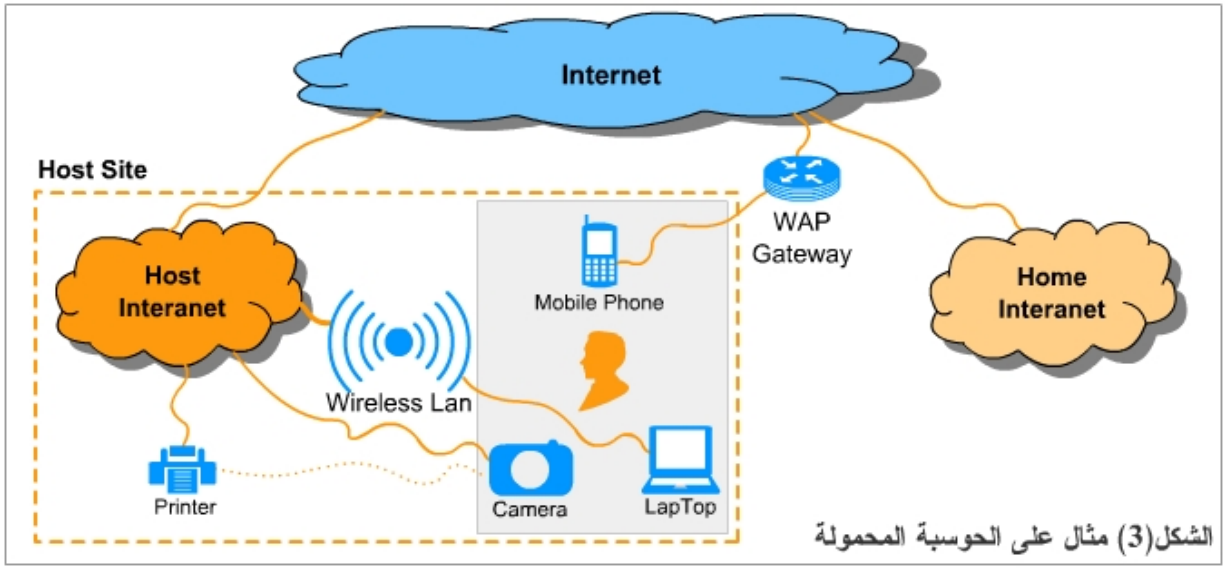
- الحواسيب المحمولة
- التجهيزات المحمولة في اليد، بما فيها المساعدات الشخصية الرقمية والهواتف النقالة وآلات التصوير الرقمي.
- التجهيزات التي يمكن ارتداؤها، كالساعات الذكية التي لديها وظائف تشبه تلك التي للمساعد الشخصي
- التجهيزات المدمجة ضمن الأجهزة الأخرى كالغسالات وآلات التوزيع وأنظمة

الصوت والسيارات والبرادات.

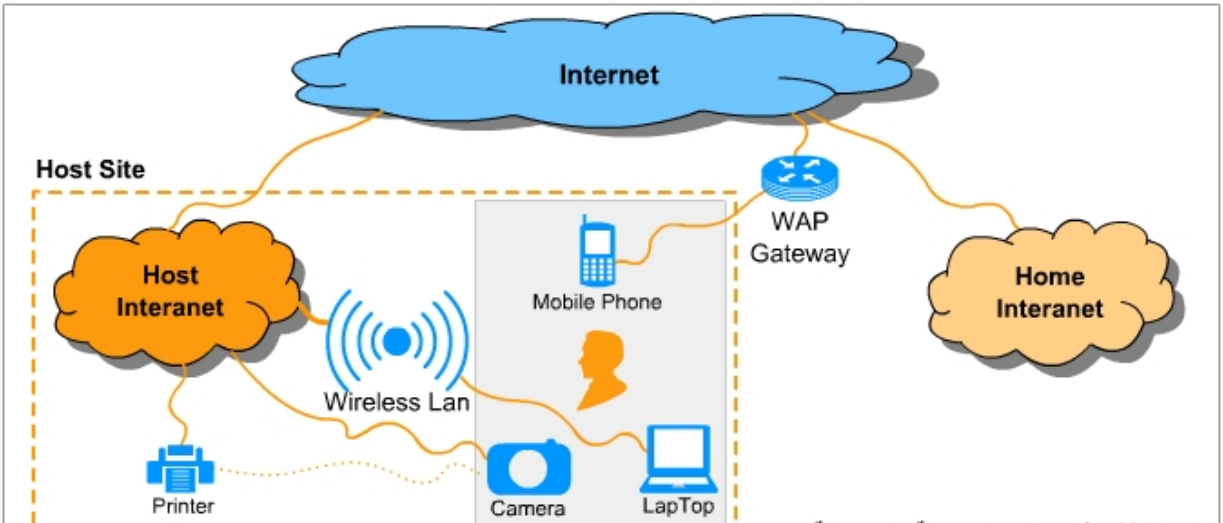
إن قابلية العديد من هذه التجهيزات للحمل والتنقل، بالإضافة إلى قدرتها على الاتصال بسهولة بالشبكات في مواقع مختلفة، يجعل من الحوسبة النقالة أمراً ممكناً. ويقصد بالحوسبة النقالة القيام بالمهام أثناء تنقل المستخدم أو تواجده في بيئة غير بيئته المعتادة. ففي الحوسبة النقالة، يستمر المستخدمون البعيدون عن شبكتهم الأساسية (أو شبكة الموطن الخاصة بهم) في الحصول على إمكانية الوصول إلى الموارد باستخدام التجهيزات التي يحملونها معهم. ويمكن للمستخدمين الاستمرار في تصفح الإنترنت، واستخدام الموارد المتاحة لهم في شبكتهم الأساسية، لا بل هناك توجه يتنامى بسرعة نحو تمكين المستخدمين من استخدام موارد متاحة لهم في المواقع التي يتنقلون ضمنها، ويعرف هذا التوجه بالحوسبة العارفة للموضع **location aware computing**.

وقد ظهر مؤخراً شكل جديد من أشكال الحوسبة ويعرف باسم الحوسبة الكلية الوجود **ubiquitous computing**. وهذه الحوسبة تعتمد على مبدأ يختلف كلياً عن نموذج الحاسب المكتبي الذي كان يفرض على المستخدم.

يظهر الشكل 3 مستخدماً متنقلاً يزور منظمة أخرى (نسميها بالمنظمة المضيفة **host organization**). ويظهر الشكل شبكة الإنترانت الأساسية للمستخدم بالإضافة إلى شبكةالإنترانت الموضوعة في الموقع الذي يزوره المستخدم. وتتصل كلتا الشبكتين مع شبكة الإنترنت .



الشكل(3) مثال على الحوسبة المحمولة



الشكل(3) مثال على الحوسبة المحمولة

لدى المستخدم ثلاثة أشكال متاحة من الاتصال اللاسلكي:

- اتصال مع الشبكة اللاسلكية المحلية للشبكة المضيفة باستخدام الحاسب المحمول. وهو متاح لمسافة بضعة مئات من الأمتار (ولنقل أنه طابق من طوابق المبنى)، ومنه إلى شبكة الإنترنت المضيفة عبر بوابة تربط الشبكة اللاسلكية المحلية بالشبكة المضيفة.
- اتصال باستخدام الجهاز الخلوي المتصل مع الإنترنت باستخدام بروتوكول الواب **WAP**.
- اتصال بالأشعة تحت الحمراء من جهاز التصوير الرقمي الخاص به.

ويمكن للمستخدم القيام بالعديد من المهام باستخدام هذه الاتصالات، كالحصول على معلومات من الإنترنت باستخدام جهازه الخلوي وتحميلها على جهاز الحاسب المحمول باستخدام اتصال **Bluetooth**، وبالتالي طباعتها بالاستفادة من تجهيزات الشبكة الداخلية المضيفة.

تطرح الحوسبة المحمولة والحاضرة في كل مكان العديد من التحديات تضاف إلى تحديات الأنظمة الموزعة. ومنها الحاجة إلى اكتشاف الموارد المتاحة في بيئة المضيف الخارجية والغريبة عن بيئة شبكة الموطن التي يعمل عليها المستخدم عادة، والعمل على جعل التجول بين شبكة الموطن والشبكات المضيفة يتم بدون الحاجة إلى إعادة برمجة الجهاز المحمول، وجعل المستخدمين يتعودون على كون الاتصال اللاسلكي المتاح أثناء التنقل محدوداً، والمحافظة على خصوصية وأمن المستخدمين أثناء تنقلهم بين الشبكات المختلفة.

4. تحديات الأنظمة الموزعة

هناك العديد من التحديات التي تقف في وجه بناء الأنظمة الموزعة، كعدم تجانس المكونات العتادية والبرمجية لهذه الأنظمة، والحاجة إلى المحافظة على انفتاح النظام بما يسمح بإضافة أو استبدال مكونات النظام، والأمن الذي يتأثر كثيراً بوجود الشبكات الحاسوبية، وقابلية التوسع التي تسمح بالتكيف مع تزايد عدد مستخدمي النظام بدون التأثير على الأداء، ومعالجة الأخطاء التي يصعب اكتشافها وتحديدها، ومعالجة التوارد والحاجة إلى التنسيق بين المكونات المتباعدة، والشفافية التي يحتاجها مطورو التطبيقات ليكونوا قادرين على بناء تطبيقاتهم بدون فرضيات متعلقة بتوضع المكونات والموارد.

عدم التجانس Heterogeneity: إذ يمكن أن يتضمن النظام الموزع العديد من الشبكات وأنظمة التشغيل والعتاديات والتطبيقات البرمجية المختلفة. وتعتمد الأنظمة الموزعة على بروتوكولات الشبكات لمعالجة الفروقات بين الشبكات، وقد أصبح هذا الموضوع أقل تعقيداً بعد أن أصبحت بروتوكولات الإنترنت معياراً على أرض الواقع لشبكات المعطيات. كما تم تطوير عدد من البرمجيات الوسيطة **middleware** لحل مشكلات عدم التجانس الأخرى مثل مشكلة ترتيب الثمانيات في تكوين العدد الصحيح.

الانفتاح Openness: إذ يجب أن تكون النظم الموزعة منفتحة وقابلة للتوسع لتضم منظومات جديدة. والحد الأدنى من الانفتاح هو نشر الواجهة البرمجية التي يجب على مطوري المكونات الالتزام بها للتمكن من مكاملة برمجياتهم مع النظام الموزع.

الأمن security: إن استخدام الشبكات كوسيلة حيوية لنقل المعطيات والنفاذ إلى الموارد المشتركة يشكل نقطة ضعف حقيقية في أمن النظام الموزع. ويمكن اعتماد التشفير كأداة أساسية لتأمين حماية الموارد المشتركة والحفاظ على سرية المعلومات الحساسة عند نقلها عبر الشبكة ولكن يبقى هناك نوع من الهجومات لا توجد له حلول جذرية حتى الآن وهي الهجومات من نوع منع الخدمة **Denial of service**، والمقصود بها أن الهجوم لايهدف إلى تخريب النظام وإنما إلى منعه من العمل، وعادة ما يكون ذلك عبر إغراق النظام بملايين الطلبات الخلبية مما يؤدي إلى عدم قدرته على تلبية الطلبات الحقيقية.

قابلية التوسع scalability: يعتبر النظام الموزع قابلاً للتوسع في حال كانت كلفة إضافة مستخدم جديد ثابتة (الكلفة = كمية الموارد التي يجب إضافتها). يجب إذا أن تصمم الخوارزميات المستخدمة للنفاذ إلى البيانات المشتركة بحيث لا ينتج عنها نقاط اختناق. كما يجب

أن تتم هيكلة البيانات لتكون هرمية بما يضمن أسرع وقت للنفاذ، ويمكن أن تنسخ البيانات المستخدمة بكثرة على عدة مواقع بهدف تسريع الوصول إليها.

معالجة الأعطال: باعتبار أن أي إجرائية أو حاسوب أو شبكة قابلة للتعطل باستقلالية كاملة عن باقي مكونات النظام، فمن الضروري أن يكون كل مكون على علم بحالات العطل التي يمكن أن تحصل في المكونات الأخرى التي يعتمد عليها في عمله. بالإضافة إلى كونه مصمماً ليكون قادراً بالتعاطي مع هذه الأخطاء.

التوارد concurrency: إن وجود عدة مستخدمين في آن واحد ضمن نظام موزع يطرح إشكالية النفاذ المتوارد إلى الموارد المختلفة. ويجب بالتالي أن تصمم الموارد بحيث تكون قادرة على العمل بأمان في بيئة متواردة. والمشكلة الأساسية هنا تكمن في كون السواد الأعظم من الخوارزميات التي سبق ورأيناها سابقاً (مادة نظم التشغيل) تعتمد على وجود ذاكرة مشتركة (مثال: السيمافور) وبالتالي لا يمكن البتة استخدامها في بيئة موزعة. هناك إذا حاجة إلى خوارزميات تعتمد على استخدام الرسائل لحل مشكلة التوارد.

5. تطبيقات الأنظمة الموزعة واستخداماتها

تستخدم الأنظمة الموزعة في العديد من التطبيقات المتنوعة، وندرج فيما يلي بعض التطبيقات التي يفضل فيها الاعتماد على نظام موزع عوضاً عن نظام تقليدي.

1. التطبيقات التفرعية وعالية الأداء: يمكن تشغيل التطبيقات التفرعية على الحواسب متعددة المعالجات التي تستخدم الذاكرة الفيزيائية المشتركة. ولكن هذه الأنظمة غير قابلة للتوسع بسهولة، وبالتالي يفضل استخدام الأنظمة الموزعة فهي أكثر قدرة على التوسع لتشمل أعداد كبيرة من وحدات التشغيل. علماً بأننا لا نقصد هنا الحواسيب الخاصة بالحوسبة الفائقة الأداء، فتلك الحواسيب لها بنيان حاسوبي خاص وأنظمة تشغيل خاصة.
2. التطبيقات المقاومة للأعطال: إن الأنظمة الموزعة هي أكثر وثوقية من الأنظمة العادية نظراً لاستقلالية عناصر المعالجة، إذ لايؤثر حصول عطل في إحدى الوحدات أو الموارد البرمجية أو العتادية على عمل المكونات الأخرى.
3. التطبيقات الموزعة بطبيعتها: فهناك العديد من التطبيقات التي تكون موزعة بطبيعتها, ومن أهم الأمثلة عليها المناقلات المصرفية وبريمجات جافا المستخدمة في الإنترنت. ويحدد أداء هذه التطبيقات بغزارة التنفيذ (عدد المناقلات في الثانية) عوضاً عن زمن التنفيذ.
4. سمحت الأنظمة الموزعة بظهور جيل جديد من التطبيقات التي تعرف باسم التطبيقات التعاونية أو تطبيقات المجموعات **groupware**، وهي تطبيقات تسمح للمستخدمين بالعمل بشكل تعاوني. ومثال عليها هو التحرير الإلكتروني لوثيقة مقسمة إلى عدة أجزاء يكون كل مستخدم مكلفاً بجزء منها. مثال آخر هو تطبيقات المؤتمر المرئي**video conference** ، والتعليم عن بعد.

6. الأنظمة الموزعة المعيارية

مع انتشار التطبيقات الموزعة والرغبة الواضحة لدى المستخدمين ومط ـ وري التطبيقات في الاستفادة من هذه التقانة، ظهرت الحاجة إلى ضمان إمكانية التعامل المتبادل **Interoperability** بين مختلف التطبيقات وبيئات التطوير والتنفيذ المستخدمة. ولضمان هذه الإمكانية، هناك حاجة إلى وجود عدد من المعايير أو الأنظمة المعيارية التي تكون ضماناً للتوافق. ونورد فيما يلي وصفاً لأهم هذه الأنظمة المعيارية:

بيئة DCE من منظمة الأنظمة المفتوحة **OSF**، وتتضمن العديد من التقانات المعيارية للحوسبة الموزعة. وتقدم هذه البيئة خدمات الأمن للتحكم بالنفاذ إلى البيانات وحمايتها، وخدمات الأسماء التي تسهل العثور على الموارد الموزعة، بالإضافة إلى نموذج قابل للتوسع بسهولة لتنظيم عدد كبير ومتوزع على نطاق واسع من المستخدمين والخدمات والبيانات وقد صممت هذه البيئة لتكون قادرة على العمل في وسط غير متجانس من المكونات العتادية والبرمجية. وتتضمن بيئة **DCE** التقانات التالية التي سنشرح العديد منها بالتفصيل في الفصول القادمة:

- النياسب **threads**.
- الاستدعاء البعيد للإجراءات **RPC: Remote procedure call**.
- خدمة الزمن الموزعة **Distributed Time service**
- خدمة الأسماء **Name service**
- خدمات الأمن **Security services**
- الخدمات الموزعة **Distributed services**

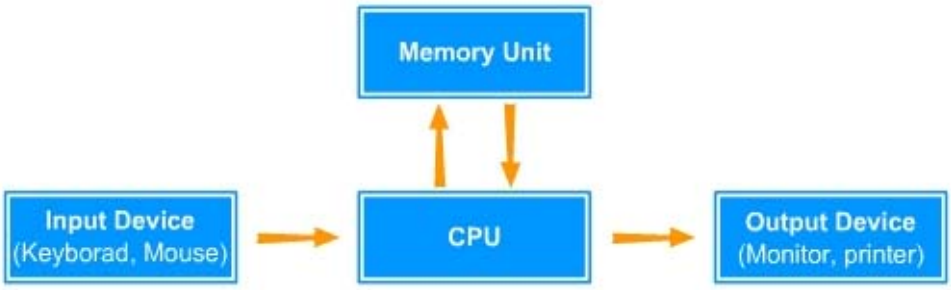
هناك أيضاً معيار **CORBA** الذي قامت بتطويره مجموعة **OMG**، وهو بنيان معياري يعتمد على النموذج الغرضي التوجه لتحقيق طلبات الخدمات بشفافية. أما ميكروسوفت فلديها النموذج **DCOM: Distributed Component Object Model**، وتعتمد شركة **SUN** على بيئة **JavaBeans** المبنية على لغة جافا.

الفصل الثاني

النماذج المختلفة للتوزيع، توزيع الموارد والبيانات والتحكم

1. تذكير بالبنيان الحاسوبي

يتكون الحاسوب بحسب نموذج فون نيومان **Von Neumann** من وحدة معالجة مركزية **CPU**، ووحدة ذاكرة **memory unit**، ووحدة الدخل والخرج **I/O**. وحدة المعالجة المركزية هي دماغ الحاسوب، فهي تنفذ التعليمات المخزنة في الذاكرة عن طريق جلب التعليمات من الذاكرة إلى المعالج وتنفيذها واحدة تلو الأخرى. أما وحدة الذاكرة فهي مسؤولة عن تخزين التعليمات (ويعرف البرنامج على أنه سلسلة من التعليمات) والمعطيات التي تتعامل معها هذه التعليمات. وتكون وحدة الدخل والخرج مسؤولة عن إدخال وإخراج التعليمات والمعطيات من وإلى المعالجات.

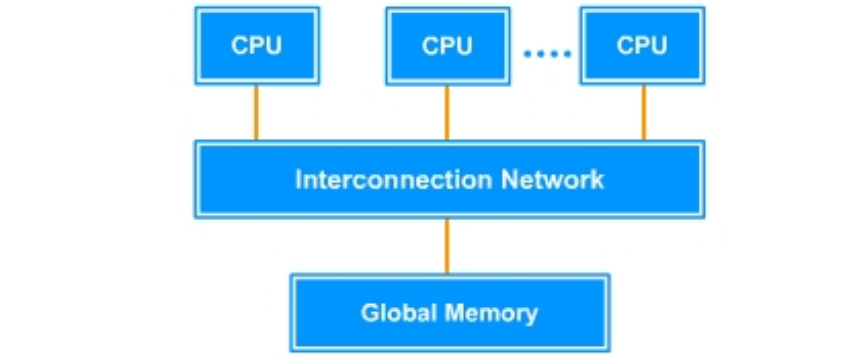


الشكل (1) مكونات الحاسوب بحسب نموذج فون نيومان Von Neumann

فإذا افترضنا أنه بالإمكان مضاعفة وحدة المعالجة والذاكرة فإنه من الممكن الوصول إلى شكلين من التنظيم الحاسوبي:

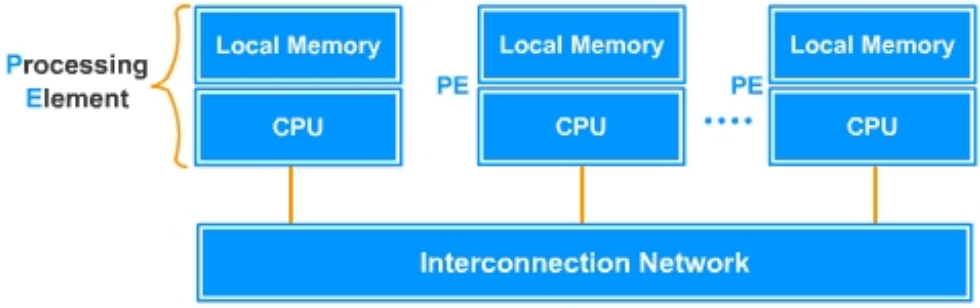
1. بنية تعتمد على ذاكرة مشتركة كما يظهرها الشكل **2-a**، يكون فيها لوحدات المعالجة فضاء عنونة موحد مشترك. وتعرف هذه الأنظمة تحت اسم الأنظمة شديدة الترابط **tightly coupled system**. ويكون التخاطب بين وحدات المعالجة المختلفة عبر الذاكرة المشتركة وباستخدام علميات القراءة والكتابة التقليدية
2. بنية تعتمد على ذاكرة موزعة فيزيائياً كما يظهرها الشكل **2-b**. وفي هذه البنية ترتبط كل واحدة معالجة بالذاكرة المحلية الخاصة بها. ويسمى الزوج (وحدة المعالجة والذاكرة) بعنصر المعالجة **Processing Element** أو بكل بساطة بالمعالج. وتعرف هذه الأنظمة تحت اسم الأنظمة قليلة الترابط **loosely coupled systems**. وفي حالة نظام مبني على الذاكرة الموزع، تتم الاتصالات بين المعالجات عبر شبكة ربط تقدم للمعالج المرسل عملية **send** وللمعالج المستقبل عملية **receive**. ويظهر الشكل **2** شبكة الربط على أنها مستخدمة لربط الوحدات المختلفة للنظام ولدعم تنقل البيانات والتعليمات. ولا

تظهر وحدة الدخل والخرج في الشكل، ولكن لا يعني هذا أنها أقل أهمية.



(a)

الشكل (2–a): الأنظمة شديدة الترابط tightly coupled system



(b)

الشكل (2–b): الأنظمة قليلة الترابط loosely coupled systems

الشكل (2) شكلان أساسيان للتنظيم الحاسوبي متعدد المعالجات

ولا يرتبط اختيار نموذج الاتصال بالضرورة مع الشكل الفيزيائي للنظام، إذ يمكن التمييز بين التشارك الفيزيائي والذي تكون فيه المشاركة على مستوى العتاديات، والتشارك المنطقي الذي يمثله نموذج البرمجة. ويظهر الشكل 3 أربع حالات مختلفة للتشارك.

	Logically Shared	Logically distributed
Physically Shared	Shared Memory	Simulated Message Passing
Physically distributed	Distributed Shared Memory	Message Passing

الشكل **3** التشارك المنطقي في الذاكرة يقابله التشارك الفيزيائي

تمثل العلبة المسماة بالذاكرة المشتركة **shared memory** البرمجة وحيدة المعالج والمتواردة المبنية على نموذج الذاكرة المشتركة، والتي تحتاج إلى وجود تنسيق بين الإجرائيات **processes** للتأكد من أن المعطيات المشتركة تحافظ على اتساقها. وتشمل آليات التزامن المستخدمة للتنسيق بين الإجرائيات المتواردة تلك التي رأيناها باستفاضة في مقرر نظم التشغيل، كالسيمافور **semaphore** والمراقب **monitor** والمقطع الحرج **critical section** والحواجز **barriers**.

أما العلبة المسماة بنقل الرسائل **message passing** فتمثل الأنظمة المبنية على الذاكرة الموزعة فيزيائياً والتي يتم فيها الاتصال باستخدام الرسائل، وبالذات تعليمتي الارسال **send** والاستقبال **receive**. ويمكن محاكاة نقل الرسائل **simulated message passing** للحصول على نموذج برمجي منطقي يعطي الانطباع بأن الذاكرة موزعة بينما هي فعلياً مشتركة، وفي هذه الحالة ترسل الرسائل عبر خزانات **buffers** مشتركة في الذاكرة.

وفي النهاية نجد العلبة المسماة بالذاكرة المشتركة الموزعة **distributed shared memory (DSM)**، وتعرف أيضاً باسم الذاكرة الافتراضية المشتركة الموزعة **distributed shared virtual memory** أو نظام الذاكرة الموزعة **distributed memory system**، ويستمد هذا النموذج أهميته من كونه يحاول الاستفادة من السهولة للبرمجة باستخدام الذاكرة المشتركة عن طريق إعطاء المبرمجين الانطباع بأن لديهم ذاكرة مشتركة وبالتالي يمكنهم تعميم التطبيقات المتواردة التي تعودوا على برمجتها سابقاً، وتلافي البرمجة باستخدام الرسائل التي تتضمن العديد من المشكلات واحتمالات الخطأ.

2. التعريف التفصيلي للأنظمة الموزعة

عند الحديث عن الأنظمة الموزعة نواجه العديد من المفردات التي تقترن بهذا المفهوم بشكل عام، مثل: موزعة **distributed**، شبكة **network**، تفرعي **parallel**، متوارد **concurrent**، لا مركزي **decentralized**. ونظراً لكون الأنظمة الموزعة مفهوماً جديداً إلى حد ما، فلا يوجد تعريف نهائي متفق عليه، وإن كانت هناك عدة تعاريف مقبولة (كالتعريف الذي أوردناه في الفصل الماضي). بالمقارنة مع الحوسبة التسلسلية، فإن الحوسبة التفرعية والمتواردة والموزعة تشمل العمل الجماعي والمنسق لمجموعة من عناصر المعالجة. وبالتالي فإنه من الممكن أن يكون لدى القارئ بعض الالتباس حول المعنى الفعلي لهذه المفاهيم. من الضروري إذاً قبل التعمق في الأنظمة الموزعة بيان الفروقات بين هذه المفاهيم التي تتقاطع بين بينها البعض.

تشمل الحوسبة التفرعية **parallel** إمكانية التعامل مع مجموعة من البيانات باستخدام واحدة تنفيذ واحدة. وفي الحواسيب التفرعية من النوع **SIMD (Single-instruction, multiple-data)** بإمكان عدة وحدات معالجة تنفيذ عملية واحدة على مجموعة من عناصر البيانات في آن واحد.

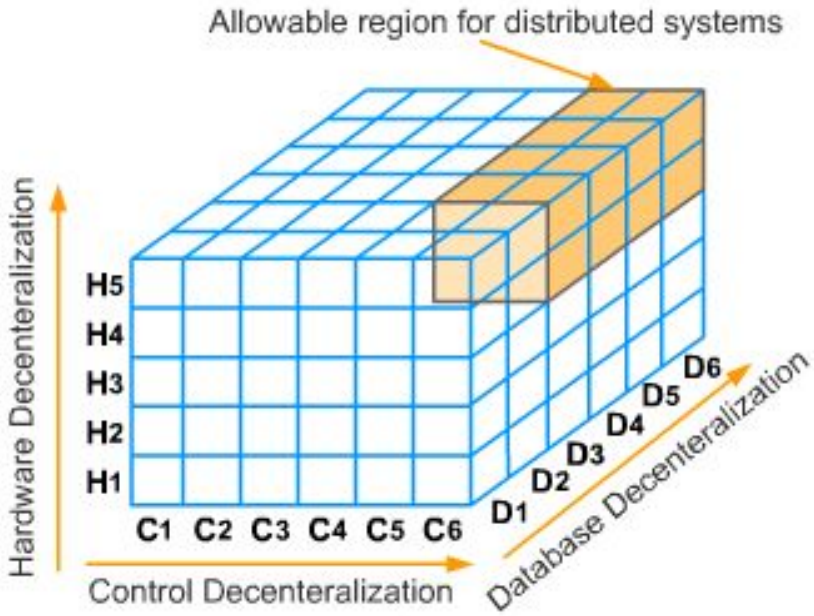
يقصد بالحوسبة المتواردة إمكانة تنفيذ عدة عمليات بترتيب عشوائي.

في حالة الحوسبة الموزعة يكون أداء الحوسبة متعلقاً إلى حد كبير بسرعة وسائط الاتصال التي تصل بين وحدات المعالجة المختلفة، والمستخدمة لنقل البيانات ورسائل التحكم.

يقال عن نظام أنه مركزي **centralized** في حال تجمع مكوناته كلها في موقع واحد، ويقال عنه أنه لامركزي **decentralized** في حال توضع مكوناته في مواقع مختلفة. وفي حال عدم وجود تنسيق بين المكونات المختلفة لنظام لامركزي يقال عنه أنه شبكي **networked**؛ أما عندما يكون هناك تنسيق كبير بين هذه المكونات يقال أن النظام موزع. ومن أهم النماذج التي تعطي تعريفاً للأنظمة الموزعة نموذج إنسلو **Enslow** الذي يقترح أن النظام الموزع يتضمن ثلاثة أوجه أو أبعاد للتوزيع: توزيع التجهيزات أو العتاديات، وتوزيع التحكم، وتوزيع البيانات.

Distributed system = distributed hardware + distributed control + distributed data

وبالإضافة إلى ذلك، فإن التعريف يفترض أيضاً أن توزيع الموارد يجب أن يكون شفافاً بالنسبة إلى التطبيقات والمستخدمين. ويظهر الشكل 4 الصيغة العامة لنموذج إنسلو. حيث يمكن لنظام أن يصنف على أنه موزع في حال كانت الأجزاء الثلاثة: التجهيزات، والتحكم، والبيانات قد وصلت إلى درجة معينة من اللامركزية.



الشكل (4) نموذج إنسلو للتوزيع

ويعرف نموذج إنسلو خمسة سويات لتوزيع التجهيزات تتراوح بين **H1** إلى **H5**:

- **H1**: وحدة معالجة واحدة مع وحدة تحكم داخلية واحدة (الحديث هنا عن وحدة التحكم داخل المعالج، ولا يجوز الخلط بينها وبين التحكم بمعنى نظام التشغيل).
- **H2**: وحدة معالجة واحدة مع عدة وحدات معالجة رياضية ومنطقية **ALU**. وتوجد وحدة تحكم واحدة.
- **H3**: تفصل الوحدات الوظيفية المتخصصة، مثل وحدة معالجة ومعالج مساعد لحسابات الفاصلة العائمة.
- **H4**: تعدد المعالجات مع وجود منظومة دخل وخرج واحدة وذاكرة واحدة مشتركة بين المعالجات كافة.
- **H5**: تعدد الحواسيب مع وجود معالجات متعددة ومنظومات دخل وخرج متعددة وذاكرة محلية في كل حاسوب.

وعلى نفس المنوال يعرف النموذج ستة سويات لتوزع التحكم تزداد مع زيادة اللاتوزع، وهي تتراوح من **C1** إلى **C5**، ومعرفة على الشكل التالي:

- **C1**: مركز تحكم واحد وثابت، ونلاحظ أن النظام لا يحتوي بالضرورة على عدة وحدات معالجة
- **C2**: مركز تحكم واحد ومتغير. وفي حال وجود عدة وحدات معالجة يتنقل مركز التحكم عبر الزمن من وحدة معالجة إلى أخرى.
- **C3**: بنيان سيد/عبد ثابت. وعلى سبيل المثال، في حال كون النظام يحتوي على وحدة معالجة واحدة ومعالج مساعد، تكون وحدة المعالجة هي السيد والمعالج المساعد هو العبد.
- **C4**: بنيان سيد/عبد متغير. وعلى سبيل المثال، يمكن تغيير العبد برمجياً.
- **C5**: عدة مراكز تحكم متجانسة تستخدم فيها نسخ متماثلة من وحدة التحكم نفسها.
- **C6**: عدة مراكز تحكم غير متجانسة تستخدم فيها نسخ مختلفة من وحدة التحكم.

أما على صعيد المعطيات، فالموضوع أكثر تعقيداً. إذ يمكن تمييز نوعين من البيانات التي يمكن توزيعها: الملفات **files** والأدلة **directories** التي تحوي هذه الملفات. ويمكن للتوزيع أن يأخذ أحد شكلين: النسخ **replication** حيث تكون هناك عدة نسخ من نفس البيانات في أماكن عدة، والتجزئة **partition** حيث تقسم البيانات إلى عدة أجزاء ويوضع كل جزء منها في أحد المواقع. وبناء عليه فإن التصنيف على مستوى البيانات يكون على الشكل التالي:

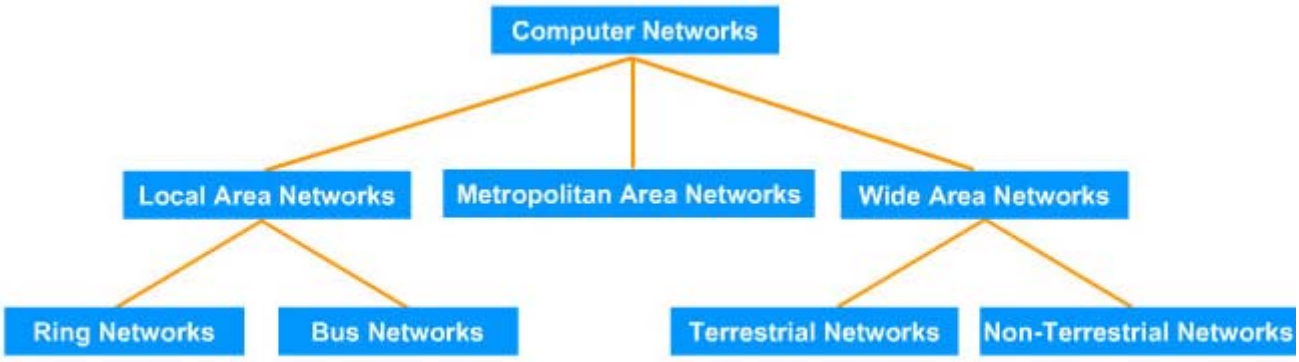
- **D1**: قاعدة بيانات مركزية تحتوي على نسخة واحدة من الملفات والأدلة
- **D2**: تكون الملفات موزعة مع دليل مركزي واحد لاتوجد منه نسخة محلية أو جزء محلي
- **D3**: قاعدة بيانات متعددة النسخ **replicated** مع وجود نسخة من الملفات والدليل في كل موقع
- **D4**: قاعدة بيانات مجزأة **Partitioned**، مع وجود نسخة أساسة **master** تحتوي على نسخة مضاعفة كاملة من كل الملفات.
- **D5**: قاعدة بيانات مجزأة، مع وجود نسخة أساسية تحتوي على نسخة مضاعفة فقط من الدليل.
- **D6**: قاعدة بيانات مجزأة، ولاتوجد وجود نسخة أساسية من الملفات أو من الدليل.

وبتحليل الشكل رقم 4، يبدو بوضوح أن نموذج إنسلو يعتبر أنه يمكن إطلاق اسم نظام موزع على أي نظام يحقق المتطلبات التالية:

- سوية توزيع التجهيزات هي **H4** أو أعلى.
- سوية توزيع البيانات هي **D3** أو أعلى.
- سوية توزيع التحكم هي **C5** أو أعلى.

ويعطي شريدر نموذجاً بديلاً للنظام الموزع، يعتمد على مبدأ الأعراض **symptoms** والاحتمالية، فإذا كان النظام لديه كافة الأعراض المدرجة أدناه فمن المحتمل أن يكون نظاماً موزعاً:

- تعدد عناصر المعالجة
- وجود تجهيزات خاصة للربط البيني بين عناصر المعالجة
- الفشل المستقل لعناصر المعالجة
- وجود حالة مشتركة بين العناصر

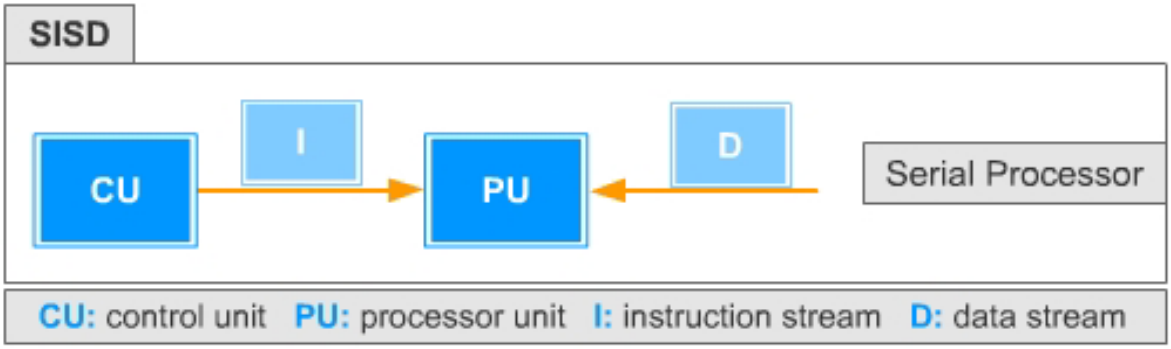


الشكل (5) تصنيف الشبكات الحاسوبية

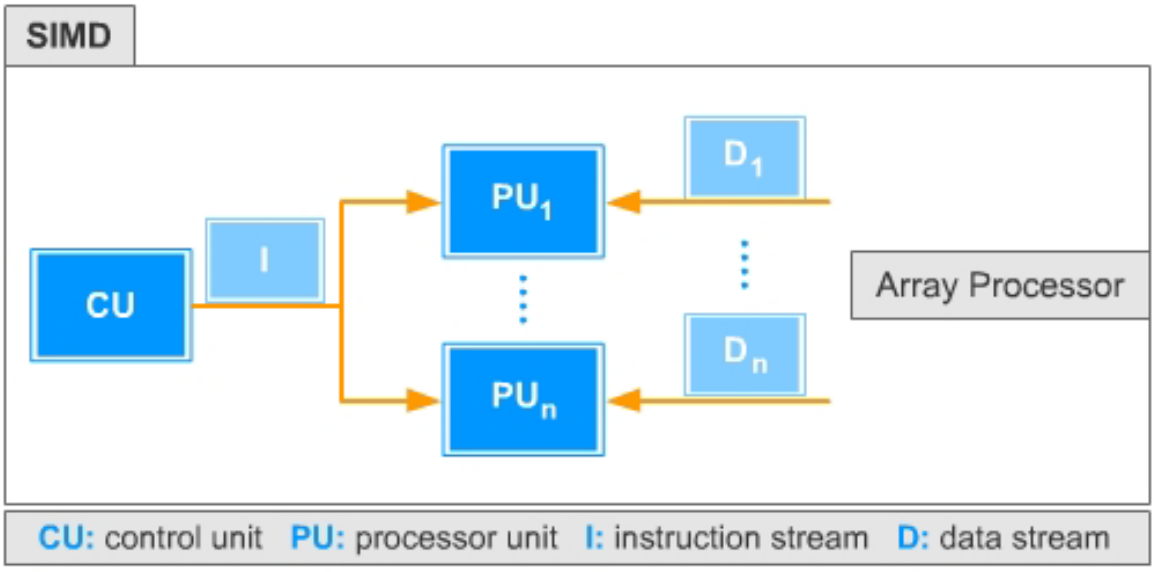
لقد أدرج العديد من الباحثين شبكات الحواسيب والحواسيب التفرعية كجزء من الأنظمة الموزعة. ويظهر الشكلان **5** و**6** تصنيفاً للشبكات وللحواسيب التفرعية. وهناك نماذج لبنيان الحاسوب لا تعتمد على نموذج فون نيومان، ومنها نموذج تدفق البيانات **data flow** المبني على التقييم الجشع **greedy evaluation** حيث يتم تنفيذ تابع مكون من عدد من التعليمات عندما تكون بيانات الدخل الخاصة بها جاهزة. أما نموذج الاختصار **reduction model** فهو مبني على التقييم الكسول، أي لا يتم تنفيذ التابع إلا عند الحاجة إلى خرجه.

وأغلب الحواسيب التفرعية مبنية على بنيان فون نيومان، أما النموذج الأكثر استخداماً لتصنيف الأنظمة المبنية باستخدام بنيان فون نيومان فهو نموذج فلين **Flynn** الذي يعتمد تعدد مسالك البيانات والتعليمات:

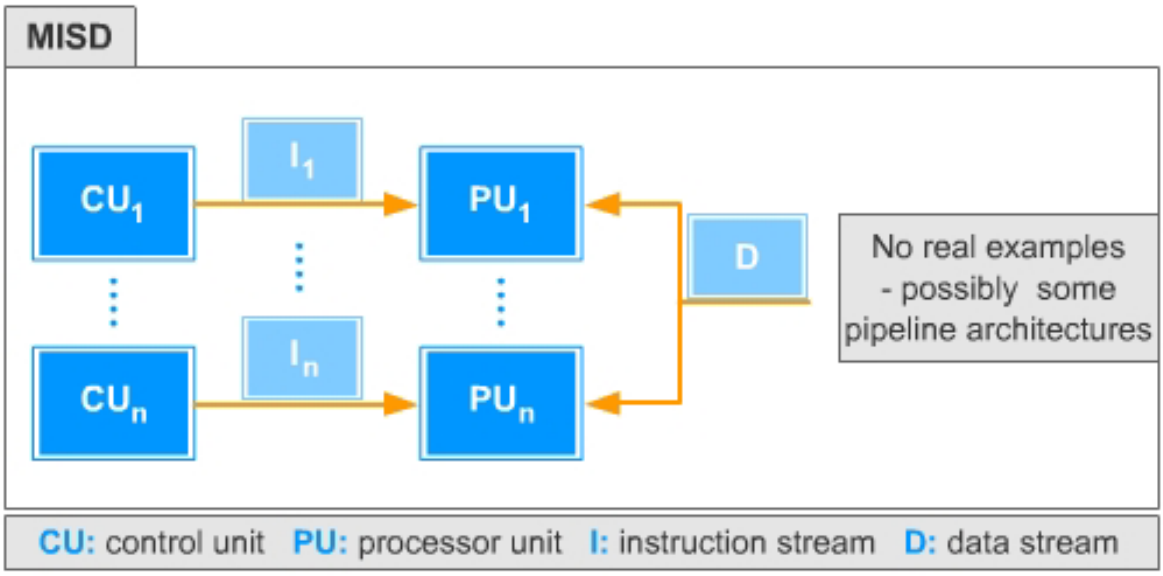
- **Single Instruction Single Data (SISD):** وهو البنيان وحيد المعالج التقليدي، ويمكن أن يتضمن على آليات شبه تفرعية لزيادة السرعة مثل خط الإنتاج **pipeline** ومضاعفة أجزاء وحدة المعالجة والدخل والخرج.



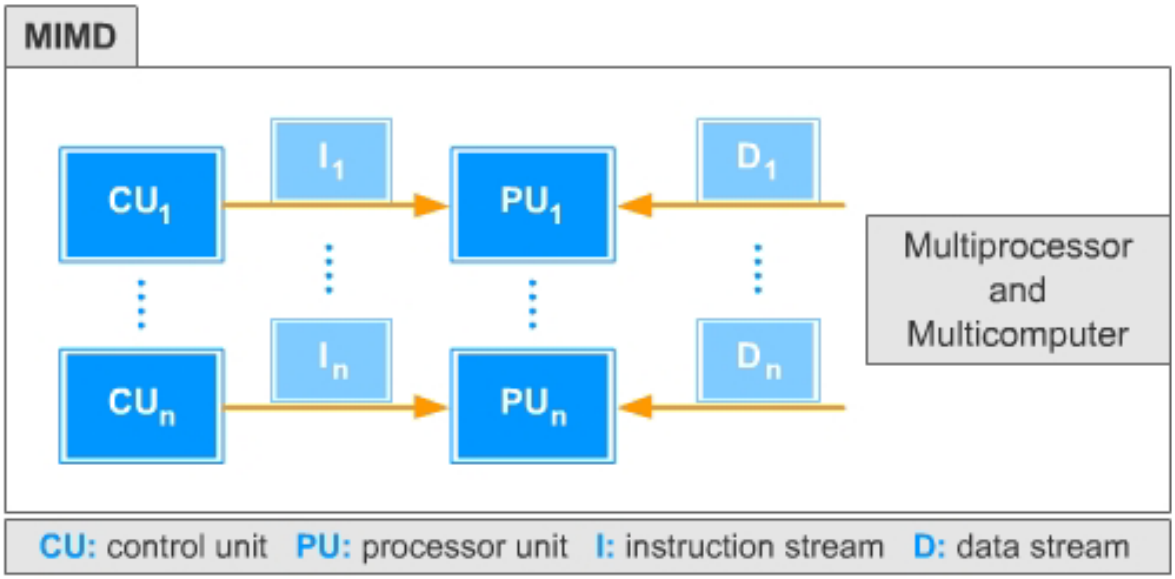
- **Single Instruction Multiple Data (SIMD):** في هذه الحالة يقوم كل معالج بتنفيذ نفس التعليمة في نفس الوقت على مجموعة البيانات الخاصة به. ومن أهم الأمثلة عليها مصفوفات المعالجات التي تستخدم في معالجة الصورة، والحواسيب الاختلاجية **systolic**.

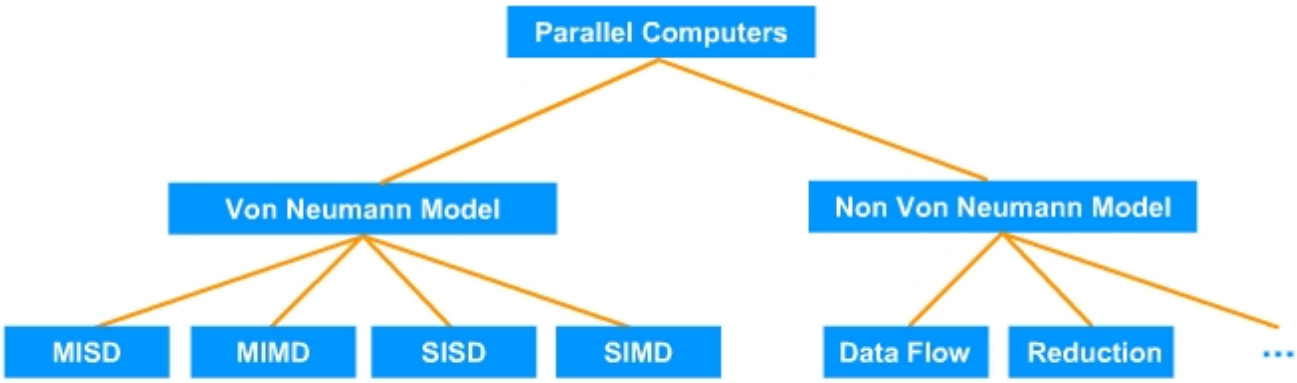


- **Multiple Instruction Single Data (MISD):** ويشمل هذا التنظيم عدة معالجات يتم التحكم بكل منها عبر وحدة تحكم مستقلة، وتعمل جميعها على نفس البيانات.



- **Multiple Instructions Multiple Data (MIMD):** في هذه الحالة يقوم كل معالج بتنفيذ تعليمات مختلفة على مجموعة بيانات الخاصة به، وهي الحالة التي تنضوي تحتها أغلب الأنظمة الموزعة.





الشكل (6) تصنيف الحواسيب التفرعية

تشمل التصنيفات المذكورة في الأشكال **5** و**6** معظم الأنماط الهامة للبنيان الحاسوبي التي سبق وأوردتها الأدبيات الفنية. وستتم مناقشة التمييز المتعلق بالربط في المقطع القادم. ولكن هناك العديد من الخيارات التصميمية الأساسية التي لم تؤخذ بعين الاعتبار في هذه التصنيفات:

- التفرع ذو الحبة الناعمة **fine grain** أو الخشنة **coarse**.
- الاتصال بالذاكرة المشتركة مقابل الرسائل
- الأنظمة ذات الاستخدام العام مقابل الأنظمة ذات الاستخدام الخاص
- الذاكرة الشاملة مقابل الذاكرة المحلية
- التحكم المركزي مقابل التحكم الموزع
- النفاذ المتماثل مقابل النفاذ غير المتماثل
- الاتصال المباشر مقابل الاتصال غير المباشر

التفرع ذو الحبة الناعمة هو شكل من أشكال الحوسبة التفرعية يكون فيه حجم الحوسبة لكل إجرائية صغيراً بالمقارنة مع كلفة الاتصال. وبالمقابل، يكون حجم الحوسبة لكل إجرائية في التفرع ذي الحبة الخشنة أكبر عدة مرات من كلفة الاتصال.

الأنظمة ذات الاستخدام العام هي أنظمة تستخدم في تطبيقات عامة، وإلا فهي أنظمة ذات استخدام خاص. وتعتبر الأنظمة من نوع **MIMD** ذات استخدام عام، أما أنظمة **SIMD** فهي مناسبة للتطبيقات الرقمية التي تطبق فيها نفس العمليات على مجموعة كبيرة من البيانات، كمرشحات الصور مثلاً.

يمكن النفاذ إلى الذاكرة الشاملة **Global memory** من أي معالج، بينما يقتصر في حالة الذاكرة المحلية على المعالج المرتبط بها. وفي كثير من الأحيان ترتبط الذاكرة المحلية مع الذاكرة الشاملة باستخدام ذاكرة خائبة **Cache** تشكل ذاكرة عالية السرعة ومنخفضة العطالة.

يجري تبادل المعلومات في الأنظمة ذات الذاكرة المشتركة عبر قراءة وتعديل متحولات مخزنة في الذاكرة المشتركة. أما في الأنظمة المبنية على تبادل الرسائل فإن الشكل الطبيعي لتبادل المعلومات هو إرسال الرسائل واستقبالها.

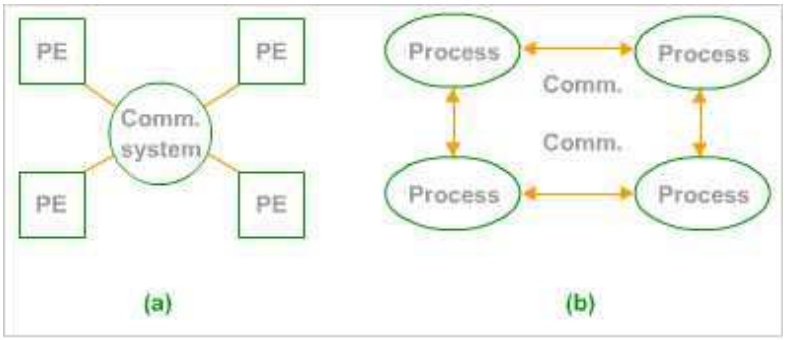
في حالة النفاذ المتماثل إلى الذاكرة تكون كافة أجزاء الذاكرة متوضعة على مسافة متساوية من كافة المعالجات. وبالنتيجة يكون زمن النفاذ لأي عنوان في الذاكرة من أي معالج ثابتاً تقريباً. ولكن هذه النمط من المنظومات يعاني من صعوبة التوسع، نظراً لاعتماده على بنية ناقل **bus** مشترك لوصل المعالجات مع الذاكرة المشتركة، ويتحول هذا الناقل إلى نقطة اختناق مع تزايد عدد المعالجات. ويمكن وضع ذاكرة خابئة ضمن المعالج للحد من التنافس على الناقل، ولكن تظهر هنا مشكلة اتساق الذاكرة الخابئة ومطابقة محتواها لمحتوى الذاكرة المشتركة. وتحل الأنظمة ذات النفاذ غير المتماثل هذه المشكلة عبر تزويد كل معالج بذاكرة محلية وبوصلة إلى شبكة ربط بيني تستخدم بالوصول إلى الذاكرة الخاصة بالمعالجات الأخرى، وهذه الشبكة قابلة للتوسع. وتتغير أزمنة النفاذ إلى الذاكرة بحسب موقع البيانات.

تتكون شبكة الربط البيني من عدد من خطوط الاتصال والنواقل والمبدلات التي تصل المعالجات ببعضها البعض في النظام الموزع أو التفرعي. وفي حالة الربط المباشر تكون المعالجات مرتبطة ببعضها البعض عبر عدد من الوصلات من نوعة نقطة-لنقطة **point-to-point**. أما في الاتصالات غير المباشرة، فإن المعالجات تتصل عبر مبدلات، وعادة ما يكون هذا النوع من الوصل مرناً وقابلاً للتغير بسهولة عبر تغيير برمجة المبدلات.

3. النموذج المعتمد للنظام الموزع

سنستخدم في باقي المقرر التعريف التالي للأنظمة الموزعة:

النظام الموزع هو نظام يبدو كأنه نظام عادي بالنسبة للمستخدمين ولكنه يعمل على مجموعة من عناصر المعالجة ذات الاستقلالية. ولكل عنصر معالجة فضاء ذاكرة فيزيائية خاص به، ويكون الزمان اللازم لنقل الرسائل غير مهمل. ويكون التعاون وثيقاً بين هذه العناصر، ويجب أن يدعم النظام عدداً عشوائياً من الإجرائيات وتوسيعاً ديناميكياً لعناصر المعالجة.



الشكل (7) الشكل الفيزيائي والمنطقي لنظام موزع

ويظهر الشكل 7 الشكل الفيزيائي والمنطقي لنظام من هذا النوع. ويجب أن يحقق النظام الموزع الخصائص التالية:

- عدد عشوائي من الإجرائيات، وتسمى كل إجرائية بالموارد المنطقية **logical resource**.
- عدد عشوائي من عناصر المعالجة. ويسمى أيضاً بالموارد الفيزيائية **physical resource**.
- الاتصال عن طريق الرسائل. ويؤمن ذلك نقل الرسائل باستخدام طريقة تعاونية.
- الإجرائيات تعاونية. بمعنى أن التفاعلات بين الإجرائيات تتم بشكل تعاوني، بحيث تتعاون الإجرائيات في حل مسألة مشتركة عوضاً عن وجود عدة تطبيقات مستقلة لا علاقة لها ببعضها البعض.
- زمن الاتصال: يكون زمن الاتصال بين عنصري معالجة غير مهمل.
- مقاومة الأعطال بحيث لا يوجد أي عنصر سواء فيزيائياً كان أو منطقياً يؤدى تعطله إلى توقف النظام بأكمله.
- الانحدار الهادئ **graceful degradation**: إذ يجب أن يكون لدى النظام الإمكانات

لإعادة تشكيل النظام أو إعادة توزيع الموارد لمجابهة الأعطال. والاستمرار في العمل ولو بسوية أدنى من التقليدية.

بناء على هذا التعريف لايمكن اعتبار الشبكات الحاسوبية بأشكالها المختلفة **(LAN, MAN, WAN)** أنظمة موزعة. وذلك لأن الإجرائيات في المواقع المختلفة لا تعمل سوية بشكل تعاوني.

لا تعتبر أيضاً الذاكرة المشتركة فيزيائياً شكلاً من أشكال النظام الموزع نظراً لعدم وجود استقلالية عن الأعطال. بالمقابل، فإن الذاكرة الموزعة المشتركة لديها خاصة الاستقلال عن الأعطال، وهي قادرة على تحقيق انحدار هادئ. وبالتالي تعامل الأنظمة المبنية على الذاكرة الموزعة المشتركة على أنها نوع خاص من الأنظمة الموزعة.

وفي نفس السياق (التمييز بين الأنظمة الموزعة والتي لا يمكن اعتباره أنظمة موزعة) يدرج تانينبوم **Tanenbaum** عدة محددات أساسية لثلاثة أنواع مختلفة من الأنظمة: الشبكات الحاسوبية، الأنظمة الموزعة، والأنظمة متعددة المعالجات. وتتضمن تلك المحددات دعم المعالج الأحادي الافتراضي **virtual uniprocessor**، عدد النسخ من نظام التشغيل، وعدد أرتال التشغيل، ودلالة التشارك في الملفات **file sharing semantics**. ويمكن تلخيص وضع هذه المحددات بالنسبة للأنظمة المختلفة المذكورة على الشكل التالي:

- لاتدعم الأنظمة الشبكية عادة المعالج الافتراضي الأحادي، بينما تدعمه الأنظمة الموزعة ومتعددة المعالجات.
- تدعم الأنظمة الشبكية تعدد أنظمة التشغيل بينما تستخدم الأنظمة الموزعة ومتعددة المعالجات نظام تشغيل واحد، وإن كانت هناك عدة نسخ من نفس النظام في حالة النظام الموزع مقابل نسخة واحدة في حالة تعدد المعالجات.
- لدى الأنظمة الموزعة والشبكية عدة أرتال للتنفيذ بينما هناك رتل واحد في نظام متعدد المعالجات
- إن تعريف التشارك في الملفات واضح في الأنظمة الموزعة ومتعددة المعالجات. أما في حالة الأنظمة الشبكية فهو غير واضح ويحتاج إلى العديد من الفرضيات.

الفصل الثالث

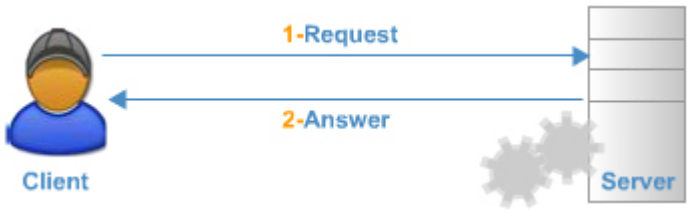
المخدم الزبون

1. المبدأ التصميمي للمخدم/الزبون

ظهرت بيئة المخدم/الزبون **client-server** في نهاية الثمانينيات من القرن الماضي، وكانت في ذلك الوقت تشكل بداية الحوسبة الموزعة. وقد حققت نجاحاً كبيراً يستمر حتى يومنا هذا، إذ اعتمد هذا النموذج لدى الكثير من الشركات المصنعة لأنظمة التشغيل والتجهيزات.

الفكرة الأساسية في نموذج المخدم الزبون هو نمذجة نظام التشغيل على شكل مجموعة من الإجرائيات المتعاونة التي نسميها مخدمات، وتقدم خدمات إلى المستخدمين الذين نسميهم زبائن. وتعمل هذه الإجرائيات فوق نظام تشغيل خاص يدعم العديد من الوظائف التي تسهل الاتصال بين الإجرائيات كما سنرى لاحقاً. تعمل إجرائيات المخدم والزبون عادة في فضاء المستخدم (وليس في فضاء النواة)، ويمكن أن يتضمن الحاسب الواحد إجرائية واحدة، أو عدة زبائن، أو عدة مخدمات، أو مزيجاً من الاثنين.

يعتمد نموذج المخدم-الزبون في عمله على مبدأ بسيط جداً: يرسل الزبون طلباً إلى المخدم (مثال: أعطني جزءاً من ملف)، ويقوم المخدم بتنفيذ الطلب وإعادة النتيجة إلى الزبون أو يعيد رسالة خطأ في حال عدم تمكنه من تقديم الخدمة.



تتم عمليه التخاطب بين المخدم والزبون عبر إرسال رسائل بين الإجرائيتين وتعتمد الرسائل عادة على بروتوكول اتصالات غير متصل بهدف تلافي الكلفة الكبيرة للاتصال المتصل، لاسيما عندما يكون المخدم والزبون جزءاً من شبكة محلية واحدة، وبالتالي فإن خطر حصول ضياع أو مشكلة في الرسائل المنقولة قليل جداً. وتقدم نواة نظام التشغيل استدعاءات للنظام خاصة بإرسال واستقبال الرسائل، غالباً ما تقتصر على استدعائين اثنين لتبسيط الأمور:

- إرسال باستخدام الاستدعاء: send(dest, &mptr)، وذلك لإرسال الرسالة التي عنوانها **mptr** إلى الإجرائية التي هويتها **dest**، وتبقى الإجرائية المرسلة متوقفة حتى يتم إرسال الرسالة.
- استقبال باستخدام الاستدعاء: receive(addr, &mptr)، وينتج عنها توقف الإجرائية المستقبلة إلى حين وصول الرسالة. وعند وصول الرسالة تنسخ في العنوان الذي يشير

إليه **mptr** وتتابع الإجرائية المستقبلة عملها.

هذا الشكل للاستدعاءات هو طبعاً الشكل العام، وسنتطرق إلى الأشكال المختلفة لاحقاً في هذا الفصل.

2. مثال تطبيقي أولي على الزبون والمخدم

يظهر المثال في الشكل رقم **1** مثالاً عن الشكل العام لمخدم ملفات وزبون للمخدم بلغة **C**.

```
/* Definitions needed by clients and servers                    */
#define TRUE                    1
#define MAX_PATH                255     /* maximum length of file name          */
#define BUF_SIZE                1024    /* how much data to transfer at once    */
#define       FILE_SERVER               243     /* file server's network address    */

/* Definitions of the allowed operations */
#define       CEATE                     1       /* create a new file            */
#define READ                            2       /* read data from a file and return it
        */
#define WRITE                   3       /* write data to a file                 */
#define       DELETE                    4       /* delete an existing file      */

/* Error codes */
#define       OK                        0       /* operation performed correctly */
#define       E_BAD_OPCODE              -1      /* unknown operation requested  */
#define E_BAD_PARAM             -2      /*error in a parameter                  */
#define       E_IO                      -3      /* disk error or other I/O error */

/* Definitions of the message format */
struct message {
        long source;                    /* sender's identity                    */
        long dest;                      /* receiver's identity                  */
        long opcode;                    /* requested operation                  */

        long count;                     /* number of bytes to transfer          */

        long offset;                    /* position in file to start I/O        */
        long result;                    /* result of the operation              */
        char name[MAX_PATH];            /* name of file being operated on       */
        char data[BUS_SIZE];            /* data to be read or written           */
};
```

الشكل **(1)** ملف الترويسة المستخدم لتوصيف المخدم والزبون

ويمكن تلخيص النقاط الأساسية كما يلي:

- يستخدم كلا الزبون والمخدم توصيفاً مشتركاً لعديد من بنى المعطيات(لاسيما الرسائل) وهي كلها موضوعة في الملف **header.h**.
- يبدأ الملف بتعريف ثابتين أساسيين: **MAXPATH** والذي يحدد العدد الأقصى للمحارف في اسم الملف (متضمناً اسم الدليل) و **BUF-SIZE** الذي يحدد حجم حجم البيانات التي يمكن نقلها بين المخدم والزبون في عملية واحدة. يلي ذلك ثابت **FILE-SERVER** يعرف عنوان الشبكة للمخدم الذي يمكن للزبون إرسال الرسائل عليه.
- المجموعة اللاحقة من الثوابت تعرف أرقام ترميز العمليات المتفق عليها بين المخدم والزبون.
- تتضمن كل إجابة ترميزاً لنتيجة. إذا نجحت العملية، فإن الترميز يحتوي على الأغلب على معلومات مفيدة (كعدد المحارف التي تمت قراءتها). في حال عدم وجود قيمة للإعادة (حالة إنشاء ملف)، يعيد القيمة **OK**. أما في حال فشل العملية فيستخدم ترميز النتيجة للإعلام بسبب الفشل (***K.BAD_OPCODE, K.BADYARAM***).
- الجزء الأهم هو تعريف الرسالة بحد ذاتها. في مثالنا هي صيغة تتضمن **10** حقول، وهذه الصيغة مستخدمة في كافة الرسائل المتبادلة بين المخدم والزبون (الطلب والإجابة).

Field name	Field meaning	Possible values
Source, dest	Identify the sender and receiver	
Opcode	Operation code	*CREATE, READ, WRITE, DELETE*
Count, offset	Used for parameters	
Extra1, extra2	Provide space for additional parameters in case the server is expanded in the future	
Result	Holds result value for server-to-client replies	
Name (array)	Name of the file being Accessed	
Data (array)	holds the data sent back on a reply to READ or the data sent to the server on a WRITE	

ننظر الآن إلى برمجة المخدم والزبون كما هي في الشكل **(2)**:

```
# include <header.h>
void main(void) {
        struct message m1, m2;                  /* incoming and outgoing messages      */
        int r;                                  /* result code                         */

        while(TRUE) {                           /* server runs forever                 */
                receive(FILE_SERVER, &M1);      /* block waiting for a message         */
                switch(m1.opcode) {             /* dispatch on type of request         */
                        case CREATE:    r= do_create(&m1,&m2); break;
                        case READ:      r= do_reade(&m1,&m2); break;
                        case WRITE   :  r= do_write(&m1,&m2); break;
                        case DELETE:    r= do_delete(&m1,&m2); break;
                        defaulte     :  r=E_BAD_OPCODE;
                }
                m2.result =r;                   /* return result to client             */
                send(m1.source, &m2);           /* send reply                          */
        }
}
```

(a)

```
# include <header.h>
int copy(char *src, char *dst){         /* procedure to copy file using the server     */
        struct  message m1;             /* message buffer                              */
        long position;                  /*current file position                        */
        long client=110;                        /*client's address                    */

        initialize();                   /*prepare for execution                       */
        position =0;
        do {
                m1.opcode = READ;       /*operation is a read                         */
                m1.offset = position;   /*current position in the file                */
                m1.count = BUF_SIZE;    /* how many bytes to read                     */
                strcpy(&m1.name, src);  /* copy name of file to be read to message    */
                send(FILESERVER, &m1);  /*send the message to the file server         */
                receive(client, &m1);   /* block waiting for the reply                */

                /* Write the data just received to the destination file               */
                m1.opcode = WRITE;      /* operation is a write                       */
                m1.offset = position;   /*current position in the file                */
                m1.count = m1.result;   /* how many bytes to write                    */
                strcpy(&m1.name, dst);  /* copy name of file to be written to buf     */
                send(FILESERVER, &m1);  /*send the message to the file server         */
                receive(client, &m1);   /* block waiting for the reply                */
                position +=m1.result;   /* m1.result is number of bytes written       */
        } while(m1.result > 0);         /* iterate until done                         */
          return(m1.result>=0 ? OK : m1.result);        /* return OK or error code   */
}
```

(b)

الشكل **(2)**: مثال على المخدم **(a)**، والزبون **(b)**

برمجة المخدم:

- هناك حلقة رئيسية تبدأ باستدعاء **receive** بهدف قراءة رسالة. يعطي الوسيط الأول عنوان الزبون، والثاني خزان بيانات يمكن فيه تخزين الرسالة الواردة.
- يتوقف عمل المخدم حتى تصل الرسالة، وعند وصولها يتابع المخدم عمله بناء على قيمة **opcode** المتضمنة في الرسالة.
- من أجل قيمة لـ **opcode** يتم استدعاء إجراء مختلف مع إعطائه وسائط تتضمن الرسالة الواردة وخزان لبناء رسالة الإجابة، ويعيد الإجراء قيمة توضع في الحقل **result** من رسالة الإجابة.
- يرسل المخدم الإجابة باستخدام **send** ويعود إلى بداية الحلقة ليقرأ رسالة جديدة.

برمجة الزبون:

- نرى في المثال إجراء من جهة الزبون يقوم بنسخ الملف باستخدام المخدم. يتكون جسم الإجراء من حلقة تقرأ كتلة محارف من ملف المصدر وتقوم بنسخها في ملف النسخة، وتستمر هذه الحلقة حتى يتم الانتهاء من عملية النسخ، ويستدل على ذلك من نتيجة الإجراء **read** (صفر أو قيمة سالبة). ويهدف الجزء الأول من التعليمات داخل الحلقة إلى بناء الرسالة التي تتضمن طلب القراءة **READ** وإرسالها إلى المخدم. وعند استقبال النتيجة، يتم تنفيذ الجزء الثاني من التعليمات داخل الحلقة، وهي تقوم بإعداد البيانات التي تم جلبها من المخدم ليصار إلى إرسالها ثانية على شكل طلب للكتابة **WRITE** في ملف النسخة.

ملاحظة: التعليمات المكتوبة هي فقط لإعطاء صورة عامة ولا تدخل البتة في التفاصيل، وهناك العديد من النقاط التي تم إغفالها بهدف الاختصار ولأنها غير مهمة في هذه المرحلة من الشرح.

3. الخصائص المختلفة للرسائل في نموذج المخدم/الزبون

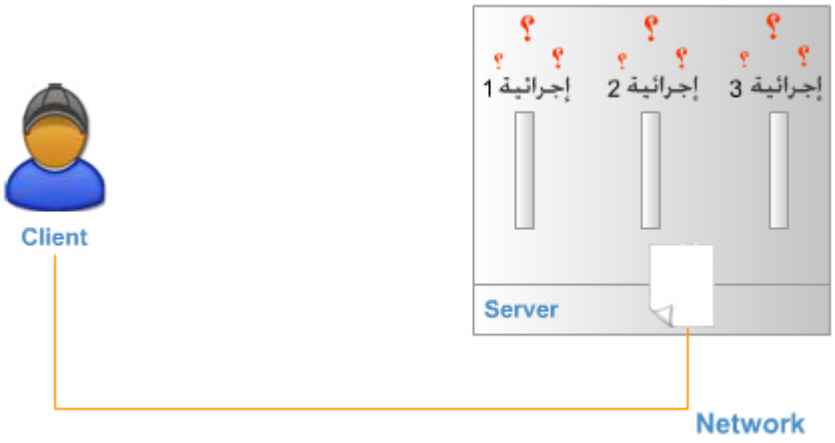
العنونة

يحتاج الزبون إلى معرفة عنوان المخدم ليتمكن من إرسال الرسالة. وفي المثال السابق كان عنوان المخدم مثبتاً في الملف **header.h** على الشكل التالي:

```
#define MAX_PATH 255          /* maximum length of a file name */
#define BUF_SIZE 1024         /* how much data to transfer at once */
#define FILE_SERVER 243       /* file server's network address */
```

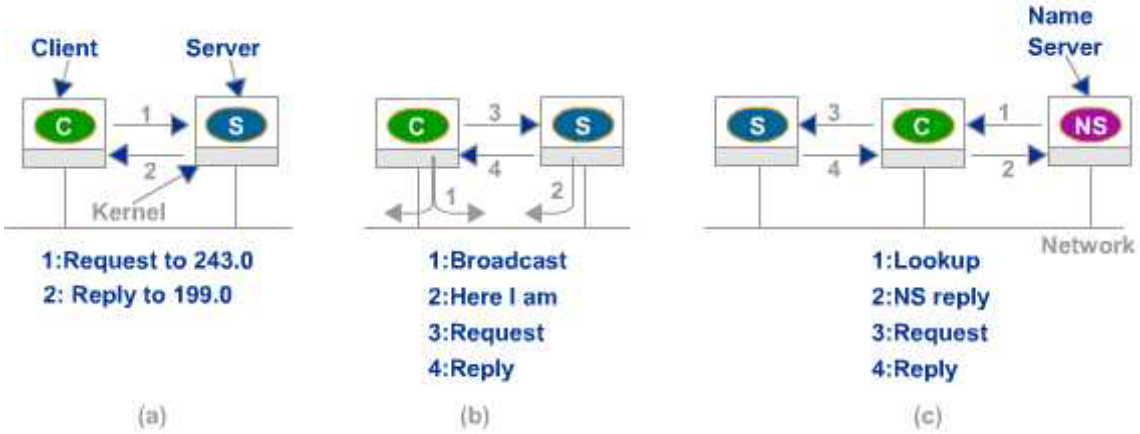
هذه الطريقة في العنونة مقبولة في الحالات البسيطة، ولكن لا يمكن الاعتماد عليها بشكل عام ويجب تأمين آلية عنونة أكثر تطوراً.

أعطينا في المثال عنواناً رقمياً للمخدم، ولكن لم نحدد ما هو معنى ذلك العنوان. يمكن للعنوان أن يوصف آلة معينة، وبالتالي يستخدم كعنوان فيزيائي من السوية 2 (يعرف أيضاً باسم **MAC Address**)، ويوضع ضمن الرسائل المرسلة. ولكن هذا لا يكفي، إذ يمكن أن تكون هناك أكثر من إجرائية تعمل على الحاسب المستخدم كمخدم، لا بل يمكن أن تكون كل من تلك الإجرائيات تقدم خدمة مختلفة عن الأخرى. يتعين بالتالي تحديد أي من تلك الإجرائيات يجب أن تستلم الرسالة وتقوم بتقديم الخدمة المطلوبة، أي أنه في حال الاقتصار على عنوان الآلة سنحصر نفسنا في حالة إجرائية واحدة على المخدم. وهو أمر غير مرغوب (وإن كان ليس بالكارثياً).



هناك آلية للعنونة تعتمد على إرسال الرسالة إلى الإجرائيات، وذلك بهدف إلغاء الالتباس الذي يمكن أن يحصل في حال وجود عدة إجرائيات على المخدم. ولكنها تدخل مشكلة جديدة، ألا وهي كيف يتم التعرف على الإجرائية. ومن أحد أكثر الآليات شيوعاً هي تقسيم العنوان إلى

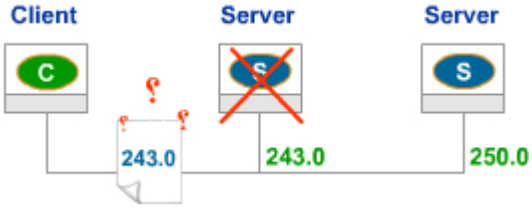
قسمين، بحيث يتضمن العنوان عنوان الحاسوب بالإضافة إلى رقم الإجرائية على الشكل ***machine.process*** (مثال: **243.4** أو **4@243** لتحديد الإجرائية رقم **4** على الحاسب الذي عنوانه **243**). وهذه الآلية مبينة في الشكل **3 (a)**.



الشكل **(3)** الخيارات المختلفة المتعلقة بعملية العنونة

هناك تنويع لهذه الآلية في العنونة تعتمد على كتابة العنوان على شكل **machine.loacl-id** عوضاً عن الشكل السابق، والحقل **local-id** هو ببساطة عبارة عن عدد صحيح عشوائي. وتستخدم الإجرائية هذا الرقم عن طريق إعلام نواة النظام بأنها ترغب في الاستماع إلى الرقم **local-id** وعندما تصل رسالة إلى الحاسب عليها العنوان**machine.localid** يقوم النظام بتسليمها إلى الإجرائية. ومن أهم الأمثلة على هذه الآلية هي المقابس **socket** التي ظهرت في البداية في نظام **Berkeley UNIX**.

وتعاني هذه الآلية من مشكلة كبيرة متعلقة بشفافية العنونة، إذ من الواضح أنه لا يمكن للمستخدم (سواء كان تطبيقاً أم شخصاً ما) أن يرسل رسالة إلا إذا كان على معرفة بعنوان الحاسوب الذي تتوضع عليه الإجرائية. فلنفترض جدلاً أن الحاسب ذا العنوان **243** المعرف أعلاه قد توقف فجأة عن العمل، وأنه يوجد هناك حاسب آخر احتياطي قادر على حل محله ورقمه **250**، فلن يكون من الممكن توجيه البيانات إليه لأنه التطبيقات كلها قد تم توليدها بناء على الرقم **243**.



خيار بديل: إعطاء كل إجرائية عنواناً وحيداً لا يتضمن رقم الحاسوب، وإحدى الطرق هي اعتماد مولد مركزي للعناوين يستخدم عداداً كعنوان للإجرائية. المشكلة هي في كون الحلول المركزية غير قابلة للتوسع بسهولة وعادة تشكل نقطة اختناق، وبالتالي لا ينصح بها البتة.

خيار آخر: يترك لكل إجرائية انتقاء عنوانها من فضاء عنونة كبير جداً (مثلاً عدد صحيح من **64** خانة)، وبالتالي يكون احتمال وجود تضارب صغيراً جداً، ولا يوجد مشكلة في توسع الحل. المشكلة هي كيف تعرف النواة المرسلة الحاسب الذي سترسل إليه الرسالة؟

لو كنا نعمل في بيئة شبكة محلية تدعم البث **broadcasting**، تقوم النواة المرسلة بإرسال رزمة من نوع **locate packet** تتضمن عنوان الوجهة. ولأنها رزمة بث أو إرسال متعدد، فستستقبل هذه الرزمة من كافة الحواسيب على الشبكة. وتقوم كافة الأنظمة المستقبلة بفحص ما إذا كانت الإجرائية المطلوبة لديها أم لا، وفي حال الإيجاب ترسل رسالة **here I am** معطية عنوان الشبكة الخاص بها. وتستعمل النواة المرسلة هذا العنوان لإرسال الرسالة إلى الإجرائية، كما تقوم بتخزين المعلومة في ذاكرة خابئة **cache** لمنع تكرار الإرسال المتعدد. وهذه الطريقة معروضة في الشكل **3 (b)**.

هذه الطريقة شفافة ولكنها تتسبب في انشغالية مرتفعة للشبكة، حتى عند وجود الذاكرة الخابئة. ويمكن تلافي هذه الانشغالية باستخدام ما يعرف **بمخدم الأسماء name server**. وآلية عمل مخدم الأسماء معروضة في الشكل **3 (c)**. وهو عبارة عن مخدم خاص معروف لدى جميع الحواسيب ضمن النظام ويحتوي على جدول يسمح بإعطاء اسم من سلسلة محارف لكل حاسوب وربط هذا الاسم مع عنوان الحاسوب. وتعتمد التطبيقات على استخدام الاسم لعنونة الإجرائيات المختلفة، وللوصول إلى العنوان انطلاقاً من الاسم تستعين الإجرائيات بمخدم الأسماء. ويتم الاحتفاظ بالنتيجة لدى ضمن ذاكرة خابئة الإجرائية الطالبة بهدف الإقلال من الطلبات المرسلة إلى مخدم الأسماء لئلا يتحول إلى نقطة اختناق. ويمكن الحد من هذه المشكلة عن طريق استخدام أكثر من مخدم للأسماء، ولكن هذا يطرح بدوره إشكالية اتساق البيانات بين المخدمات.

ويلخص الجدول التالي الخيارات المختلفة المتعلقة بعملية العنونة:

اسم الآلية	المشكلة المرتبطة بها
وضع العنوان **machine.procid** مباشرة ضمن البرنامج	انعدام الشفافية
تختار الإجرائية عنوانها ويعثر عليها عن طريق البث	ضغط على الشبكة بسبب البث
استخدام مخدم أسماء	تحول المخدم إلى نقطة اختناق

الرسائل المتزامنة وغير المتزامنة

تعتبر كافة عمليات الرسائل التي رأينا حتى الآن من النوع المتزامن **Synchronous**، أي أن الإجرائية المرسلة توضع في حالة التعليق **suspended** فور طلبها الإرسال وحتى تنتهي عملية الإرسال كلياً. وبنفس الطريقة، فإن الإجرائية المستقبلة تظل في حالة التعليق حتى يتم الانتهاء من استلام الرسالة. أي أن التعليمة التالية للارسال أو الاستقبال لا تنفذ حتى الانهاء الكلي من العملية، حتى ولو احتاج ذلك إلى ساعات.

من أهم الانتقادات للإرسال المتزامن هو كونه يحد من التوارد بدرجة كبيرة. وأحد البدائل الشائعة هو الاعتماد على الرسائل اللامتزامنة **Asynchronous**. وفي هذه الحالة فإن عملية الإرسال تتابع التنفيذ مباشرة دون أن تنتظر وصول الرسالة إلى وجهتها، اي أن إيصال الرسالة يتم على التوازي مع عمل الإجرائية المرسلة.ومن شأن ذلك رفع نسبة الاستفادة من وحدة المعالجة المركزية. إن الخيار بين الإرسال المتزامن والغير متزامن هو من مسؤولية مصمم النظام، وهناك بعض الأنظمة التي تدعم كلا الخيارين بحيث يترك الخيار للمبرمجين.

ورغم تحسن الأداء المترافق بالإرسال اللامتزامن، فإنه يعاني من مشكلة كبيرة وهي حالة عدم التعيين فيما يخصر الذاكرة التي استخدمها المرسل لإرسال الرسالة. إذ لا يجوز المساس بها طالما لم يتم ترحيل الرسالة, ولا يوجد لدى المرسل طريقة لمعرفة اللحظة التي يجوز له فيها إعادة استخدام هذه الذاكرة. وهناك عدة حلول لهذه المشكلة نلخصها كما يلي:

يتلخص الحل الأول في نسخ الرسالة ضمن الذاكرة الداخلية للنواة تقوم بعدها الإجرائية بالمتابعة وهي مطمئنة إلى إمكانية إعادة استخدام الذاكرة المخصصة للرسالة، كما هو الحال في الرسائل المتزامنة. ومن مساوئ هذا الحل الموضح في الشكل **4(b)** أنه يجري عملية نسخ إضافية لم يكن

لها داع اساساً، وبالتالي يمكن أن ينتج عنها تأخير يحد من أداء النظام.

الحل الثاني هو أكثر تعقيداً، ويتلخص في إرسال مقاطعة إلى إجرائية المرسل عندما ينتهي إرسال الرسالة لإعلامه بأن الذاكرة أصبحت متاحة للاستخدام. وإن كان هذا الحل يلغي التأخير الناجم عن النسخ، ولكن مقاطعات إجرائيات المستخدمين مزعجة وصعبة وتشكل تحدياً لا يستهان به بالنسبة للمبرمجين، بالإضافة إلى صعوبة اكتشاف الأخطاء البرمجية وتصحيحها.

ويمكن تلخيص خيارات الإرسال على الشكل التالي:

1. إرسال متزامن يتوقف فيه المرسل حتى انتهاء الإرسال. وتكون وحدة المعالجة متاحة لاستخدام الإجرائيات الأخرى.
2. إرسال غير متزامن لا يتوقف فيه المرسل ويعتمد على نسخ الرسالة، وفي هذه الحالة نلاحظ وجود هدر للوقت بسبب عملية النسخ.
3. إرسال غير متزامن يعتمد على مبدأ المقاطعة، وفي هذه الحالة لا يوجد هدر ولكن البرمجة صعبة.

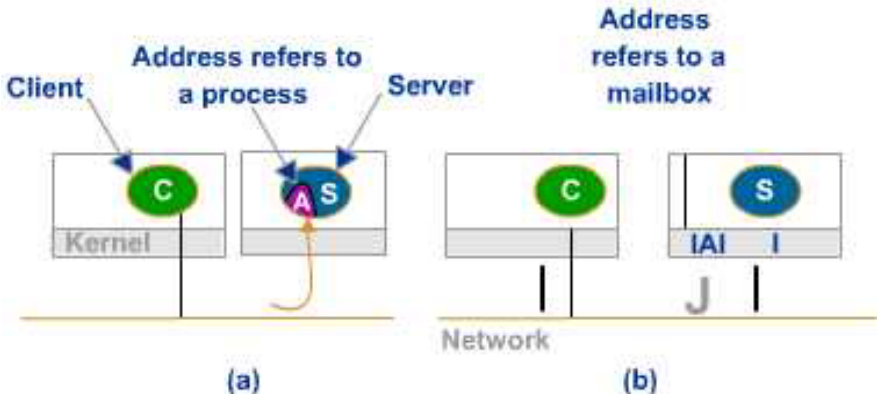
في كثير من الحالات يكون الخيار الأول هو الأفضل، ومع أنه يحد من درجة التوارد، ولكنه سهل التحقيق وسهل الفهم ويعطي ضمانات واضحة للمبرجين. إذ يعرف المبرمج أن انتهاء تعليمة الإرسال مرتبط حكماً بإرسال الرسالة. كما أنه لا يحتاج إلى حجز موارد إضافية على مستوى النواة لتخزين الرسائل. ويبقى الخيار الثاني هو الأكثر استخداماً في الحالات التي يكون فيها من المحبذ إرسال الرسائل في أقصر زمن ممكن.

ملاحظة هامة:

نحن نتكلم هنا عن مفهوم التزامن بالنسبة لمطوري أنظمة التشغيل، حيث يتركز الاهتمام على حجز موارد النواة وتحريرها. في حين أن لمطوري لغات البرمجة رأي آخر، فبالنسبة لهم لا يمكن القول بأن الإرسال متزامن إلا إذا اقترن الاستمرار بوصول الرسالة إلى وجهتها، وليس بمجرد مغادرتها الحاسوب الذي أرسلت منه.

الرسائل المخزنة

إن جميع الآليات التي رأيناها حتى الآن هي من النوع الغير مخزن، ويقصد بذلك أن نواة نظام التشغيل لا تقوم بتخزين الرسالة المرسلة في أي مكان، إذ تسلم الرسالة مباشرة إلى الإجرائية المنتظرة والتي تقوم بتنفيذ عملية **receive**، كما يظهر في الشكل **4-a**. والسؤال الذي يطرح نفسه، ماذا يحصل إذا أرسلت الرسالة بدون أن يكون هناك من ينتظرها؟



الشكل **(4)** الآليات المختلفة لتخزين الرسائل

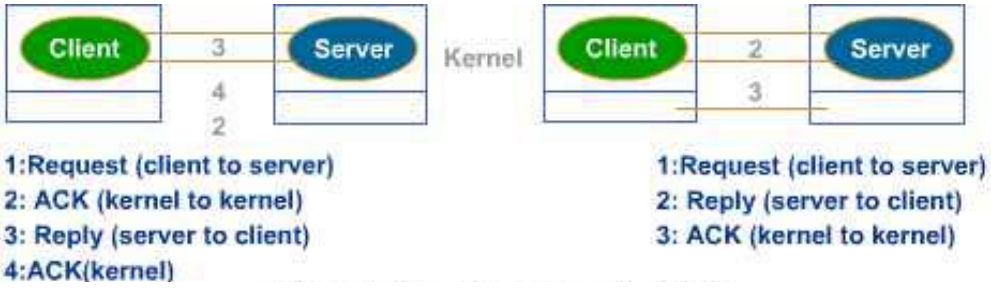
- **مقترح أول:** إهمال الرسالة، وبالتالي سيكون على الزبون بعد انتهاء زمن الانتظار أن يحاول إعادة الرسالة، على أمل أن يطلب المخدم استلام الرسالة قبل إعادة إرسالها من قبل الزبون. وهذا الحل سهل التحقيق ولكنه سيئ جداً، إذ من الممكن بعد عدة محاولات فاشلة من طرف الزبون أن يتوقف عن الإرسال ويستنتج خطأً أن المخدم يعاني من عطل أو أن العنوان خاطئ. ويمكن أن تطرأ هذه الحالة في حال وجود عدد كبير من الطلبات الواردة إلى المخدم مما لايسمح للمخدم باستقبال كافة رسائل الزبائن في الوقت المرجو.
- **مقترح ثاني:** تقوم نواة النظام التابعة للمخدمة بتخزين الرسائل الواردة لفترة ما، على أمل طلب استقبال الرسالة خلال وقت قصير. عندما تصل رسالة لم يطلبها أحد يتم تحديد زمن انتظار، فإذا انتهى ذلك الزمن دون طلب استقبال الرسالة يتم التخلص منها. وتظهر هنا مشكلة تخزين وإدارة الرسائل الواردة، إذ يجب حجز وتحرير وإدارة الخزانات اللازمة. ومن أكثر الأشكال شيوعاً هو علبة البريد. بحيث يتحتم على الأجرائية الراغبة في استقبال رسالة اخبار النواة للحصول على علبة بريد خاصة بها، ويتم توجيه كافة الرسائل الواردة إلى علبة البريد هذه كما يظهر الشكل **4-b**. وتعرف هذه الآلية باسم الرسائل المخزنة **buffered messages**. وتبقى المشكلة قائمة

في حال امتلاء علبة البريد.

وثوقية تبادل الرسائل

افترضنا حتى الآن ان عملية الإرسال موثوقة، أي أن اية رسالة مرسلة من طرف الزبون ستستقبل لدى المخدم، وفي الحقيقة فإن الأمور أعقد من ذلك. يمكن ان تضيع الرسائل، مما يؤثر على عملية إرسال الرسائل إلى حد كبير. فعندما يرسل زبون رسالة باستخدام آلية متزامنة يتم تعليق الإجرائية المرسلة إلى حين استلام الرسالة. وعندما تتابع الإجرائية عملها، لا يوجد ضمانات بأن الرسلة قد استلمت فعلياً، وبالتالي تفقد خاصة التزامن التي كنا نعتمد عليها جزءاً من أهميتها. وهناك عدة طرق للتعامل مع هذه المشكلة:

- طريقة أولى: إعادة تعريف عملية الإرسال بحيث تكون غير موثوقة، ولا يعطي النظام أية ضمانات بأن الرسالة قد تم إرسالها. وتبقى مسؤولية ضمان وصول الرسالة على الإجرائية المرسلة.
- طريقة ثانية: إرسال رسالة استلام **ACK** من نواة المستقبل إلى نواة المرسل لإعلامه بوصول الرسالة. ولا تقوم نواة النظام لدى المرسل بتحرير الإجرائية المرسلة إلا بعد وصول رسالة الاستلام. وبالتالي فإن عملية طلب خدمة تحتاج إلى أربعة رسائل (إرسال طلب، استلام الطلب، إرسال إجابة، استلام الإجابة). وهذا مشروح في الشكل **5-a**.
- الطريقة الثالثة تستفيد من وجود رسالة الإجابة **reply** التي تعيد نتيجة الطلب في المخدم وتعتبرها بمثابة استلام لرسالة الطلب **request** كما يظهر الشكل **5-b**. وتبقى رسالة الإجابة بدون استلام، ويتعلق مدى أثر هذه المشكلة على طبيعة العملية، فإذا كانت العملية بسيطة وضاعت الإجابة فسيقوم الزبون بإعادة الطلب ولن يكون هناك ضياع كبير للوقت على المخدم. إما إذا كان الطلب يحتاج إلى حسابات كثيرة، فمن المكلف أن يقوم المخدم بإعادة الحسابات من جديد بدون التحقق أولاً من عدم وصول الإجابة إلى الزبون. وتعتمد بعض الأنظمة على وجود إشعار باستلام الإجابة ترسله نواة نظام التشغيل ويبقى المخدم متوقفاً عن العمل حتى يصل هذا الإشعار. وبالتالي عندما يصل طلب جديد والمخدم متوقف يعاد إرسال الإجابة السابقة كما هي.



الشكل (5) وثوقية الرسائل المتبادلة

الرسائل المستخدمة في بناء تطبيقات المخدم / الزبون

رأينا في المقاطع السابقة أربعة خصائص أساسية للرسائل المستخدمة في النظم الموزعة بشكل عام وفي تطبيقات المخدم/الزبون بشكل خاص، وهي العنونة والتزامن والتخزين والوثوقية، ولكل منها خياراته المختلفة. ويمكن تلخيص الخيارات الأساسية ضمن الجدول المعروض أدناه، والذي يتضمن لكل خاصة ثلاثة خيارات مختلفة، وبحساب رياضي بسيط يظهر لدينا أن عدد التركيبات المختلفة هي **81** ولكن ليست كلها خيارات مناسب، والعديد منها لا معنى له.

الخاصة	الخيار الأول	الخيار الثاني	الخيار الثالث
العنونة			
التزامن			
التخزين			
الوثوقية			

الخيارات المختلفة لخصائص الرسائل

سنقوم فيما يلي بدراسة تحقيق الرسائل والبروتوكولات والبرمجيات المرافقة واللازمة لتحقيق نموذج المخدم والزبون. فمن جانب تحقيق الرسائل نجد الملاحظات التالية:

- كل الشبكات لها حجم أقصى للرسالة، أي أن الرسائل ستحتاج إلى تقسيمها إلى عدة أجزاء (رزم) ترسل منفصلة.
- يمكن لهذه الرزم أن تضيع أو تحصل فيها أخطاء أو تصل في ترتيب خاطئ، ويلجأ

مطوري البروتوكولات في هذه الحالة إلى تزويد كل رزمة برقم الرسالة وترتيب الرزمة بين الرزم الخاصة بالرسالة.

- تبقى مشكلة رسائل تأكيد الاستلام **ACK**، إذ يمكن أن ترسل لكل رزمة من رزم الرسالة مما يستهلك عرض الحزمة أو ترسل عند اكتمال رزم الرسالة مما يقعد من عملية المعالجة في حالة ضياع الرزم، لأن المرسل سيضطر لإعادة إرسال كافة الرزم. ويعتمد الخيار بشكل عام على جودة الشبكة المستخدمة، ففي حال كون الجودة مرتفعة والأخطاء قليلة نلجأ إلى الخيار الثاني.

نقطة أخرى تستحق النقاش هي على مستوى بروتوكول الرسائل المستخدم. فالشكل التالي (شكل رقم 6) يظهر ستة أنواع من الرزم المستخدمة عادة للتخاطب بين المخدم والزبون (افترضنا لتسهيل الأمور أن الرسالة صغيرة ويمكن ان تتوضع ضمن رزمة واحدة). يستخدم النوع الأول من الرزم **REQ** في إرسال طلب من الزبون إلى المخدم، اما النوع التالي **REP** فهو يستخدم في إرسال الإجابة من المخدم إلى الزبون. يلي ذلك الرزم **ACK** وهي مستخدمة في النسخ الموثوقة من البروتوكول لتأكيد استلام الرزم المرسلة.

الأنواع التالية ليست أساسية ولكنها مفيدة ونشرحها باختصار:

- رزم **AYA** ترسلها نواة نظام التشغيل لدى الزبون عند تأخر وصول إجابة المخدم، فإذا أتت إجابة من نوع **IAA**، يعرف أن المخدم لازال يعـالج الطلـب وأن الإجابـة ستصل لاحقاً. وفي حال عدم قدوم إجابة تعيد النواة الكرة عدة مرات حتـى تحـصل على إجابة أو تيأس وتعيد إجابة فشل إلى الزبون.
- يستخدم النوعان الأخيران من الرزم عندما لا يمكن قبول طلب الزبون. إذ يعيد المخدم رزمة من نوع **TA** إلى الزبون عندما يكون الطلب **REQ** مرسلاً بدون خطأ ولكن لا يمكن للمخدم استقباله في الوقت الراهن (علبة البريد ممتلئة مثلاً) فيعيد رزمة يطلب فيها من الزبون المحاولة بعد قليل. أما في حال كون الطلب **REQ** مرسلاً إلى عنوان خاطئ فيعيد الزبون رزمة **AU** للدلالة على حصول خطأ.

Code	Packet type	From	To	Description
REO	Request	Client	Server	The client wants service
REP	Reply	Server	Client	Reply from the server to the client
ACK	Ack	Either	Other	The previous packet arrived
AYA	Are you alive	Client	Server	Probe to see if the server has crashed
IAA	I am alive?	Server	Client	The server has not crashed
TA	Try again	Server	Client	The server has no room
AU	Address unknown	Server	Client	No process is using this address

الشكل (6) الأنواع المختلفة من الرزم المستخدمة في التخاطب
بين الزبون والمخدم

4. استدعاء الإجراءات عن بعد Remote Procedure Call

الهدف الأساسي من استدعاء الإجراءات عن بعد هو إعطاء مبرمجي التطبيقات الموزعة إمكانية كتابة تطبيقاتهم وكأنها تعمل بشكل مركزي، أي بدون أن يكون عليهم كتابة أية استدعاءات لعمليات دخل / خرج أو إرسال واستقبال للرسائل. والفكرة هي بسيطة جداً: السماح لإجرائية تعمل عل الحاسوب **A** باستدعاء إجراء **procedure** موجود على الحاسوب **B**، ويتم تعليق عمل الإجرائية **A** أثناء فترة تنفيذ الإجراء، ويتم نقل الوسائط إلى **B** واستعادة النتيجة تلقائياً.

من الواضح أن الفكرة الأساسية بسيطة وأنيقة، ولكن هناك العديد من المشكلات الواجب حلها، ونورد فيما يلي أهمها:

- يتم تنفيذ الإجراء المستدعي والإجراء المستدعى ضمن فضائي عنونة مختلفين.
- هناك حاجة لتمرير الوسائط والنتائج، وقد يكون ذلك الأمر معقداً، لا سيما في حال وجود اختلاف في العتاديات أو انظمة التشغيل.
- يمكن توقف أحد الحواسيب (المستدعي أو المستدعى أو كليهما)، وبالتالي يجب معالجة كافية الحالات المختلفة للتوقف.

المراحل الأساسية لاستدعاء الإجرائيات عن بعد

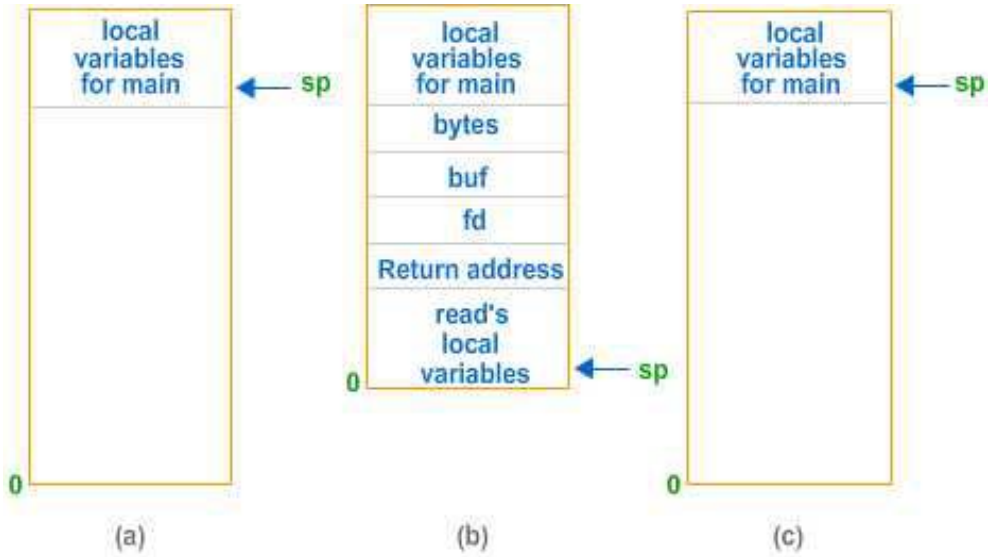
من أجل فهم آلية عمل استدعاء الإجرائيات عن بعد، من الضروري أن نفهم تماماً كيف يعمل استدعاء الإجراءات.

دعونا نعتبر الاستدعاء التالي للإجراء **read**:

```
count = read(fd, buf, nbytes);
```

حيث **fd** هو عدد صحيح، **buf** هو مصفوفة من المحارف، و **nbytes** هو أيضاً عدد صحيح.

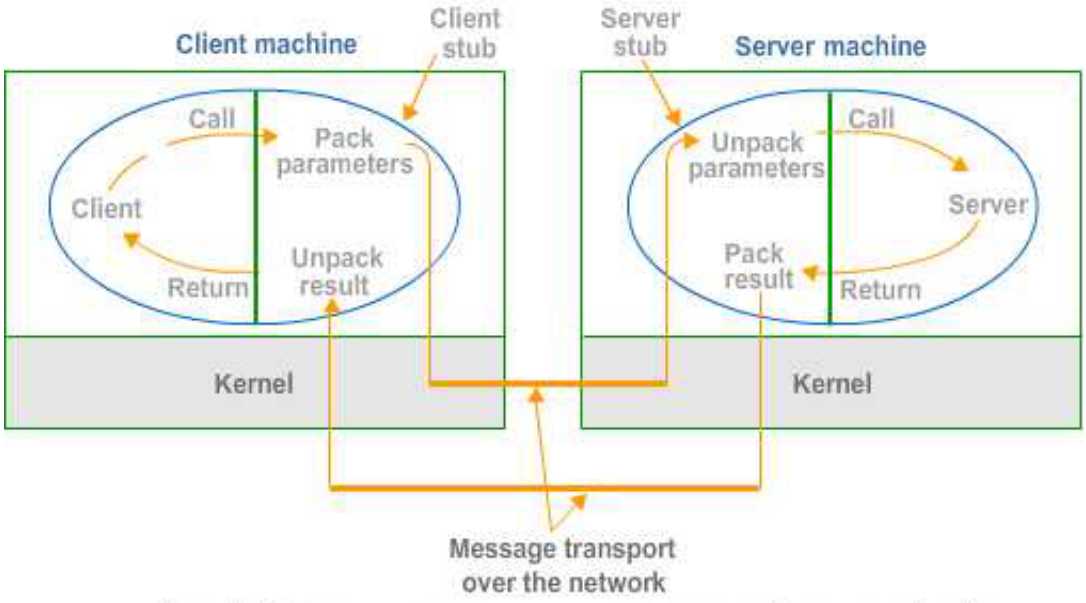
فإذا تم تنفيذ الاستدعاء محلياً، فسيكون شكل المكدس **stack** قبل الاستدعاء كماهو معروض في الشكل **7-a** لتصبح كما في الشكل **7-b** بعد أن وضعت الوسائط كلها على المكدس. وبعد الانتهاء من التنفيذ، توضع النتيجة ضمن سجل، ويحذف عنوان العودة وتجري العودة إلى سياق الطالب، والذي يقوم بدوره بحذف الوسائط من المكدس ليعود إلى حالته الأصلية كما يظهرها الشكل **7-c**.



الشكل (7) مراحل استدعاء الإجراء محلياً

تجدر بنا الآن ملاحظة النقاط التالية:

1. في لغة مثل لغة C، هناك نوعان من الوسائط: الممرة بالقيمة وهي لا تطرح أية مشكلة إذ يكتفي الإجراء بقراءتها، وحتى لو قام بإسناد قيمة لها فهذه القيمة لا تؤثر على قيمتها الأصلية لدى الإجراء الطالب. أما المتحولات الممرة بالعنوان فالهدف منها هو إعادة قيمة إلى الإجراء الطالب، وهي تحتاج بالتالي إلى معاملة خاصة. ويرتبط نوع الوسائط وطريقة تمريرها بلغة البرمجة، فلغة C تعتمد على التمرير بالقيمة في حالة الأعداد والمحارف والتمرير بالعنوان في حالة المصفوفات.
2. إن أحد أهم متطلبات الاستدعاء البعيد هي الشفافية، ويشمل ذلك المحافظة على أنواع الوسائط. فعندما يكون الإجراء بعيداً، يتم استخدام نسخة مختلفة من الإجراء عن تلك المحلية، وهي تعرف باسم الجذل **stub**، وتستخرج من مكتبة خاصة معدة للإجراءات البعيدة (عادة ما تولد هذه الإجراءات تلقائياً، انطلاقاً من توصيف صوري لترويسة الإجراءات). ويتم استدعاء هذا الإجراء بنفس طريقة استدعاء الإجراء المحلي (وبالتالي لا يوجد أي تغيير فعلياً لدى التطبيق الطالب للإجراء). والفرق هو أن هذه النسخة المختلفة تقوم بوضع الوسائط ضمن رسالة والطلب من نواة نظام التشغيل لإرسالها إلى المخدم كما نرى في الشكل رقم 8. وبعدها ينفذ الإجراء عملية **receive** وتعلق الإجرائية في انتظار وصول جواب المخدم.



الشكل (8) مراحل تنفيذ الإجراء عن بعد فيما يخص تبادل الرسائل

وتمرير وسائط الإجراء

3. لدى وصول الرسالة إلى المخدم تقوم نواة نظام التشغيل بتمريرها إلى الجذل المرتبط بالمخدم، وسيقوم باستخلاص الوسائط من الرسالة وتمريرها إلى الإجراء المطلوب من طرف المخدم بالطريقة العادية (كما في الشكل 7)، أي أن نسخ الوسائط واستعادة الجواب والوسائط الممرة بالعنوان يتم كأي إجراء عادي ولا توجد أية معاملة خاصة. فعلى سبيل المثال، في حال إجراء **read** الذي سبق وعرضناه، فإن الوسيط الثاني الثالث ممران بالعنوان إذ يحتوي الثاني على مصفوفة المحارف التي ستستقبل المحارف التي ستتم قراءتها والوسيط الثالث يمثل عدد المحارف التي تمت قراءتها فعلاً. ويقوم المخدم باستخدام متحولات داخلية ضمن الجذل تمثل كافة الوسطاء، ويمرر المتحولين المقابلين للوسيط الثاني والثالث بالعنوان، محاكياً بذلك سلوك الإجراء المحلي.
4. عند انتهاء جذل المخدم من تنفيذ الإجراء، تقوم بتحويل النتيجة إلى رسالة تعاد إلى الزبون وتتضمن كافة الوسائط الممرة بالعنوان، ونتيجة استدعاء الإجراء، وفي حالة مثالنا هذا سيتم تضمين كامل محتوى المصفوفة المستخدمة كممثل للوسيط الثاني ضمن الرسالة.
5. عند وصول الرسالة إلى الحاسوب الذي تنفذ عليه إجرائية الزبون، تقوم نواة نظام التشغيل بتحويل الرسالة إلى إجرائية الزبون. ويقوم جذل الزبون بتحويل النتيجة وتخزينها على مكدس الإجراء الطالب وكأنه إجراء محلي. وبالتالي لا يعرف فعلياً

تطبيق الزبون أن التنفيذ قد تم عن بعد ولا يوجد بالنسبة للتطبيق البرمجي أي فرق بين الإجراء المحلي والإجراء البعيد، والتمييز بينهما يتم على مستوى الجذل لكل من المخدم والزبون.

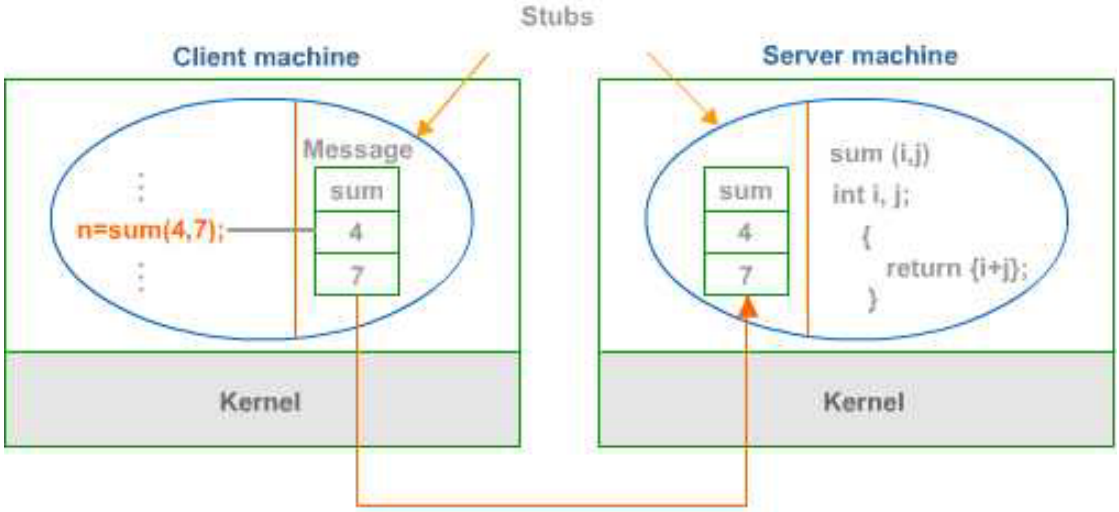
في الخلاصة نورد فيما يلي مراحل الاستدعاء باختصار:

1. يستدعي إجراء الزبون الجذل الخاص بالزبون كأي إجراء عادي.
2. يبني الجذل رسالة يطلب من نظام التشغيل إرسالها
3. يقوم نظام التشغيل بإرسال الرسالة إلى النواة البعيدة
4. تمرر النواة البعيدة الرسالة إلى الجذل الخاص بالمخدم
5. يقوم الجذل الخاص بالمخدم باستخلاص الوسائط من الرسالة واستدعاء المخدم
6. يقوم المخدم بتنفيذ الإجرائية ويعد النتيجة إلى الجذل
7. يقوم الجذل الخاص بالمخدم بتحويل النتيجة إلى رسالة ويطلب من نواة نظام التشغيل إيصالها
8. ترسل النواة البعيدة الرسالة إلى نواة الزبون
9. تعطي نواة الزبون الرسالة إلى الجذل الخاص بالزبون
10. يقوم الجذل الخاص بالزبون باسترجاع النتائج من الرسالة وتسليمها إلى إجراء الزبون.

تمرير الوسائط

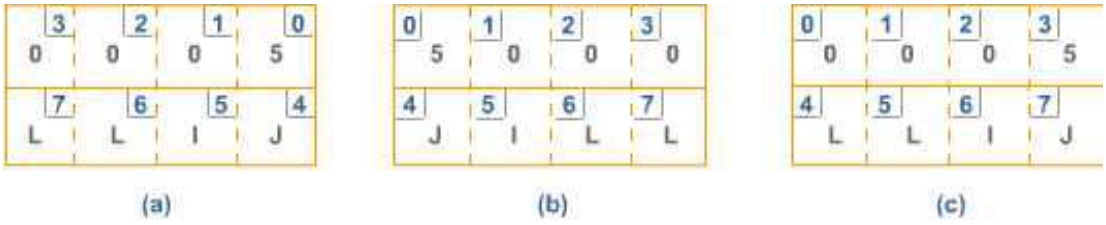
رأينا في المقطع السابق إحدى المشكلات المتعلقة بتمرير الوسائط ألا وهي مشكلة التمرير بالعنوان. ورأينا كيف يتم حلها على مستوى الجذل من طرفي الزبون والمخدم وبشكل لا يؤثر البتة على برمجة الإجراء وسلوكه المتوقع.

إن عملية تمرير الوسائط تعتمد على ترميز هذه الوسائط لتضمن ضمن رسالة يرسلها الزبون ويقوم المخدم بفك الترميز واستخلاص الرسالة. وهذه العملية بسيطة وتعمل بشكل صحيح عندما يتطابق المخدم والزبون عتادياً وبرمجياً وتكون الوسائط من أنماط بسيطة كالأعداد الصحيحة والمحارف والقيم المنطقية. وتختلف الأمور جذرياً في النظم الموزعة متوسطة الحجم والكبيرة عندما يكون هناك تعدد في أنواع الحواسيب وأنظمة التشغيل. وينتج عن ذلك اختلاف في تمثيل الأرقام والمحارف. فمثلاً لا يمكن أن يكون لدينا مخدم من الحواسيب الكبيرة لشركة **IBM** والتي تعتمد ترميز **EBCDIC** عوضاً عن **ASCII** وزبون هو حاسوب شخصي عادي يستخدم نظام **windows**، ويتم تمرير وسائط محرفية بين الإثنين بالبساطة التي يقترحها الشكل **-9-** أدناه.



الشكل **(9)** تصور أولي لعملية تنفيذ طلب الإجراء **sum (4,7)**

تظهر مشاكل أخرى شبيهة عندما يتعلق الموضوع بتمثيل الأعداد ولا سيما في موضوع التمثيل بالفاصلة العائمة **floating point** المستخدم في تمثيل الأرقام الحقيقية، وفي حالة الأعداد الصحيحة تظهر مشكلة ترتيب الثمانيات المستخدمة في تمثيل الأعداد. إذ تلجأ بعض المعالجات إلى ترتيبها من اليمين إلى اليسار كمعالجات شركة إنتل، بينما يلجأ البعض الآخر إلى ترتيبها من اليسار إلى اليمين كمعالجات شركة **sun**. ويظهر المثال في الشكل **10** المشكلة التي يمكن أن تظهر في حال اختلاف الترتيب بين المخدم والزبون. في هذا المثال هناك إجراء لديه وسيطان: عدد صحيح وسلسلة محارف من أربعة حروف، ويرمز كل وسيط باستخدام كلمة من أربع ثمانيات. يظهر الجزء **a** من الشكل كيف يمكن أن يظهر الجزء الذي يبنيه الجذل الخاص بزبون يحتوي على معالج من نوع **Intel**. تحتوي الكلمة الأولى على الوسيط الصحيح وقيمته هنا **5**، أما الوسيط الثاني فيحتوي سلسلة المحارف **JILL**.



الشكل **(10)** ترميز المتحولات وتضمينها في الرسائل

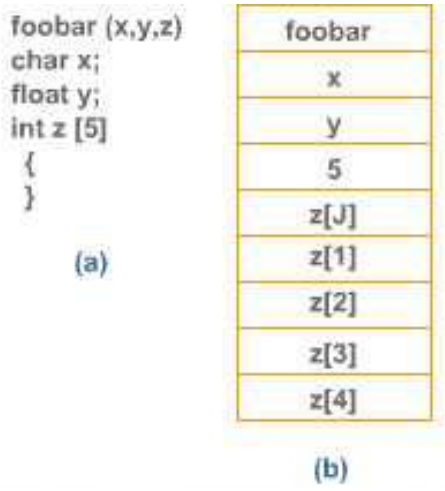
عند إرسال هذه الوسائط إلى مخدم من نوع **SPARC** ستصل الرسالة كما يظهرها الجزء **b**، أي

أن قيمة العدد الصحيح ستكون $2*2^{24}$ أي **83,886,080**، أما سلسلة المحارف فتبقى كما هي.

الحل الأفضل لهذه المشكلة هو إضافة معلومة ضمن توصيف كل وسيط تحدد نوع الوسيط وطريقة ترميزه، وذلك بحسب معيار متفق عليه بين الزبون والمخدم. وبالتالي فإن إجراء لديه **n** وسيط، سيحتاج إلى رسالة تتضمن **n+1** حقلاً، الأول يمثل الإجراء والباقون يمثلون توصيف كل وسيط.

فعلى سبيل المثال، يمكن أن نأخذ الإجراء الذي يظهره الشكل **11-a**، والمتضمن ثلاثة وسائط: محرف، عدد حقيقي، ومصفوفة من خمسة محارف.

إذا قررنا أن ننقل الحرف في الثمانية **byte** التي هي في أقصى اليمين من كلمة **WORD**، تاركين ثلاث ثمانيات غير مستخدمة، وننقل العدد الحقيقي ككلم، ومصفوفة المحارف كعدد من الكلمات مساو لعدد الخانات في المصفوفة، مسبوقة بكلمة تحتوي على عدد الخانات، كما يظهر الشكل **11-b**.



الشكل **(11)** الإجرائية والرسالة الموافقة المتضمنة وسائط الإجرائية

في حال الاتفاق المسبق على قواعد الترميز هذه، سيكون من السهل على المخدم والزبون إجراء التحويل المناسب للوسائط في الاتجاهين (ترميز، استرجاع).

يبقى الآن تحديد كيف سيتم الاتفاق. هناك خيار أول هو اعتماد صيغة معيارية لتشميل كافة الأنماط ويكون متفقاً عليها مسبقاً، مثال: الإتمام إلى **2** لتمثيل الأعداد الصحيحة، **ASCII** لتمثيل المحارف، **0** و **1** لتمثيل القيم المنطقية، **IEEE** لتمثيل الأرقام الحقيقية بالفاصلة العائمة، وتخزين كل المتحويلات بالصيغة **little endian**. من المستحيل الآن وجود أي التباس، لأن الصيغة الآن محدد ولا تتعلق بالتجهيزات المستخدمة لدى المخدم أو الزبون. المشكلة المطروحة هنا هي

الحاجة إلى تحويلات قد لا تكون (في كثير من الأحيان) ضرورية.

في مثالنا هذا، لو كان المخدم والزبون يستخدمان الصيغة **big endian** فسيتم تحويل كافة المتحولات مرتين، ومن الواضح أن في ذلك ضياعاً لكثير من الوقت. والحل هو أن يكون في بداية الرسالة عدد من الحقول تحدد بالضبط ما هي الخيارات المستخدمة في ترميز الرسالة ليقوم المستقبل بفك الترميز. وبالتالي يقوم المرسل بالترميز وفقاً لتجهيزاته ونظام التشغيل لديه، ويقوم المستقبل بفك الترميز في حال عدم تطابق خيارات الترميز التي يستخرجها من بداية الرسالة مع تجهيزاته ونظام التشغيل لديه.

من الواضح أن عملية ترميز الوسائط / النتائج وتحويلها إلى رسالة واستخلاصها من الرسالة هي عملية طويلة ولكنها نمطية وتكرارية. مما حدا بكثير من مطوري أنظمة التشغيل والتطبيقات بكتابة أدوات تسمح بالتوليد التلقائي للجذل، مثال على ذلك أداة **OIDL** من نظام **Windows**، والتي تقرأ توصيف الوسائط بلغة خاصة بذلك تعرف باسم **IDL: Interface Description Language**. ونرى فيما يلي مثالاً بسيطاً على التوصيف بلغة **IDL** للإجراءات والتي تقوم أداة **OIDL** بقراءته وتوليد ثلاثة ملفات:

- ملف الترويسة **appli.h** وهو يمثل شكل الرسالة ومعلومات الترميز المتفق عليها بين الطرفين، وهو متضمن في ملفات الجذل الخاصة بالمخدم وبالزبون.
- ملفا الجذل للمخدم والزبون: **appli_c.h** و **appli_s.h**.

```
// hello.idl: Interface specification for two procedures.
interface hello
{

void HelloProc([in, string] unsigned char * pszString);

void Shutdown(void);

}
```

يحتاج مطور التطبيقات الذي يريد الانتقال من تطبيق عادي إلى تطبيق يعمل بتقانة المخدم/ الزبون عبر تطبيق آلية الاستدعاء البعيد للإجراءات على بعض إجراءاته إلى القيام بعمليات بسيطة جداً نوردها كما يلي:

1. كتابة ملف توصيف الإجراء باستخدام لغة **IDL**، وتوليد الجذل تلقائياً باستخدام تعليمة **OIDL**.
2. كتابة برنامج للمخدم، أي الحلقة الرئيسية للمخدم والتي لا تتضمن العمليات الخاصة بالجذل، وهي عادة تكون عبارة عن نموذج جاهز يحتاج إلى بعض التعديلات الخاصة بخيارات تنفيذ المخدم (مثلاً العنوان الشبكي ورقم البوابة الشبكية التي سيستخدمها).
3. عزل أسطر تعليمات الإجراء من ملفات الزبون، ونقلها إلى ملفات المخدم كما هي بدون أي تغيير (أي لا توجد برمجة إضافية).
4. ترجمة وربط الملفات المختلفة لتوليد الملفات التنفيذية لتطبيقات المخدم والزبون.

السؤال الأخير فيما يخص تمرير الوسائط هو ماذا نفعل بالمؤشرات pointers؟

الجواب ليس بالسهل، فلا معنى للمؤشرات خارج فضاء العنونة للإجرائية التي تستخدم ضمنها، أي أن إرسال مؤشر من الزبون إلى المخدم سيفقد المؤشر معناه. وبما أن منع استخدامها غير وارد، فيجب إيجاد طريقة لاستخدامها.

الآلية المتبعة هي التالية: يستخدم المؤشر كعنوان لمصفوفة في الذاكرة، ويشير إلى أول عنصر في المصفوفة. فإذا افترضنا أن حجم المصفوفة معروف، يمكن نسخ المصفوفة ضمن الرسالة وإرسالها إلى المخدم. يقوم الجذل من طرف الزبون بتخصيص خزان في الذاكرة المؤقتة للمخدم لتخزين محتوى المصفوفة وتمرر عنوان هذه المصفوفة للإجراء.

عند انتهاء تنفيذ الإجراء ستكون النتائج قد نسخت في المساحة المؤقتة لدى المخدم. يتم إرسال رسالة تحتوي على المصفوفة المخزنة كجزء من رسالة جواب المخدم إلى الزبون، وتصل الرسالة إلى الجذل الخاص بالزبون الذي يقوم بإعادة نسخ القيم في المصفوفة الأساسية التي يشير إليها المؤشر الذي كان معطى في البداية.

معالجة الأعطال

يدخل استدعاء الإجراءات عن بعد مشكلة لم تكن موجودة في الاستدعاء المحلي للإجراءات، ألا وهي مشكلة أعطال الشبكة التي يمكن أن تحصل، وليس من السهل إخفاؤها.

نميز بين خمسة أشكال للأعطال:

1. **عدم التمكن من التخاطب مع المخدم:**

وذلك لأن المخدم متوقف أو لأن برمجيات المخدم قد عدلت أو أعيدت ترجمتها بحيث فقد التوافق بين جذل المخدم وجذل الزبون.

إن معالجة هذا العطل ليست بالبديهية، فقد يقترح أحدهم أن تعيد الإجرائية نتيجة **-1** على أنها قيمة تمثل حصول خطأ، والمشكلة أن هذه القيمة في كثير من الأحيان هي قيمة قانونية للنتيجة (مثال: **add (-2,1)** تعيد **-1**. يمكن ايضاً أن نعتمد على كتابة استثناء **exception**، ولكن هناك العديد من لغات البرمجة لا تدعم الاستثناءات، كما أن إدخالها يحتاج إلى جهد غير مهمل من المبرمجين، بالإضافة إلى أنه يعني ضمنياً انتفاء خاصة الشفافية التي بقيت ترافقنا طوال دراستنا لاستدعاء الإجرائيات عن بعد.

2. **ضياع رسالة الطلب المرسلة من الزبون إلى المخدم:**

وهي من أسهل الأعطال معالجة إذ يكفي أن يعاد إرسال رسالة الطلب في حال تأخر وصول الإجابة أو وصل استلام رسالة الطلب (في حال وجود هكذا وصل) حتى تصل الرسالة إلى وجهتها.

3. **ضياع رسالة الإجابة الواردة إلى الزبون من المخدم:**

ويبدو الحل سهلاً للوهلة الأولى، إذ يكفي أن يعيد الزبون إرسال الطلب في حال تأخر وصول الإجابة على غرار البند السابق. ولكن هناك مشكلة لها علاقة بطبيعة الإجراء المطلوب، فقد يكون سبب التأخير هو وجود بطء في المخدم وليس ضياع رسالة الجواب. إذ يمكن أن يؤدي إعادة إرسال الطلب إلى تكرار التنفيذ، وهو أمر قد لا يكون صحيحاً. فعملية قراءة قيمة ما مخزنة لدى المخدم يمكن تكرارها بدون مشكلة (توصف هذه العمليات بأنها **idempotent**)، ولكن هذا لاينطبق على عملية سحب أموال من حساب مصرفي!!

ولتلافي مثل هذه الحالات يمكن أن يكون هناك ترقيم ضمني للطلبات المرسلة بحيث تحتفظ النواة لدى المخدم برقم آخر طلب وارد من الزبون وفي حال وصول رسالة هي تكرار لطلب سابق لا يعاد التنفيذ.

4. **توقف المخدم عن العمل بعد استقبال الطلب:**

مشكلة توقف المخدم ترتبط أيضاً بإمكانية إعادة تنفيذ الإجراء، ولكنها أكثر تعقيداً من الحالة السابقة ولا يمكن حلها باستخدام ترقيم الرسائل. إذ يجب التمييز بين حالتين:

- توقف المخدم بعد الإنتهاء من تنفيذ الإجراء، وفي هذه الحالة يجب أن يعلم الزبون بذلك (عن طريق استثناء **exception** مثلاً).
- توقف المخدم قبل مباشرة تنفيذ الإجراء، وفي هذه الحالة تكون إعادة إرسال الطلب كافية.

المشكلة في عدم إمكانية التمييز بين الحالتين، فكل ما يعرفه الزبون هو أن زمن انتظار الإجابة لديه قد انتهى وعليه أن يتصرف.

هناك ثلاث منهجيات للتعامل مع هذه المشكلة:

- انتظار إعادة إقلاع المخدم وإعادة المحاولة، وتعرف هذه المنهجية بالتنفيذ مرة على الأقل **at least once**، ويمكن أن يتم التنفيذ أكثر من مرة.
- عدم إعادة المحاولة وإرسال جواب بالفشل مباشرة إلى الزبون، وتعرف هذه المنهجية بالتنفيذ مرة على الأكثر **at most once**، ويمكن ألا يكون هناك تنفيذ بالمرة.
- المنهجية الثالثة هي عدم تقديم أية ضمانات، عندما يتوقف المخدم لا يحصل الزبون على أية مساعدة أو أية وعود. ويمكن أن يتم التنفيذ بين **0** وعدد كبير من المرات، والميزة الأساسية هنا هي في سهولة التحقيق.

لكل من هذه المنهجيات الكثير من العيوب التي تعيق استخدامها. ما نتمناه هو إمكانية ضمان التنفيذ لمرة واحدة بالضبط **exactly once**، ولكن هذا التحقيق فائق الصعوبة.

إن هذا العطل (توقف المخدم) يظهر بوضوح مشكلة الاستدعاء البعيد للإجراءات، ويعتبر أحد أهم العوائق في جعل هذه الآلية شفافة كلياً للمستخدم.

5. **توقف الزبون عن العمل بعد إرسال الطلب، وقبل أن يرسل المخدم جوابه:**

تعرف هذه الحالة باسم الحوسبة اليتيمة **orphan** لأنها فقدت "والدها" أي الإجرائية التي طلبتها. المشكلة الأقل حدة هي ضياع زمن المعالجة، وبقاء عدد من الموارد المحجوزة معلقة. هناك أربع مقترحات بهذا الخصوص:

أ- **الإنهاء:** الاحتفاظ بسجل على الزبون يدون فيه الطلب المرسل قبل إرساله، وبعد إعادة إقلاع الزبون، يتم البحث ضمن السجل وإنهاء الحوسبة اليتيمة. هذا الحل غير مفيد كثيراً، إذ هناك كلفة عالية لإضافة السجل عند كل طلب لاستدعاء بعيد، بالإضافة إلى احتمال (كبير) لوجود أحفاد يتامى ناتجين عن استدعاء المخدم لإجراءات بعيدة بناء على طلب الزبون، ويستحيل تحديد موقعها وإنهاؤها.

ب- **التقمص أو الحياة الجديدة:** يخصص الزبون برقم حياة (ومنه مصطلح التقمص !!)، فعندما يعيد الإقلاع يحصل على رقم حياة يلي رقم حياته السابق، ويرسل هذا الرقم إلى كافة المخدمات. وعند استقبال هذا الرقم، تقوم المخدمات بإنهاء كافة الطلبات البعيدة (لكافة الزبائن، وذلك بسبب ظاهرة الأحفاد اليتامى). وحتى في حال عدم وصوله إلى المخدم، فإن الإجابة تحمل معها دوماً رقم حياة الزبون فإذا كان هذا الرقم من حياة سابقة يرفض الزبون استلامها

ج- **التقمص اللطيف:** وهي نسخة مخففة من الحل السابق، إذ يقوم المخدم عند استلامه لرقم حياة الزبون بالتحقق من وجود الزبائن طالبي تنفيذ الإجراءات لديه، ولا يقوم بإنهائها إلا عندما لا يتمكن من العثور على الزبون.

د- **انتهاء الوقت:** بحيث يعطى كل استدعاء بعيد حصة زمنية محددة للانتهاء من العمل، وفي حال عدم تمكنه من ذلك يجب أن يطلب تمديداً لهذه الحصة من الزبون الذي طلبه. فإذا انتظر الزبون لمدة زمنية تزيد عن هذه الحصة قبل إعادة الإقلاع، سنكون متأكدين من أن كل اليتامى قد ذهبوا. وتبقى المشكلة في اختيار قيمة مقبولة للزمن **T** في وجود متطلبات مختلفة للإستدعاء البعيد.

كل هذه الحلول غير مناسبة لأنه من الصعب جداً إنهاء الحوسبة اليتيمة بسهولة، كما ليس من السهل العثور على الموارد المحجوزة لصالحها وتحريرها.

الفصل الرابع

التزامن في الأنظمة الموزعة

1. مقدمة

رأينا في الفصول السابقة كيف يتم الاتصال بين الإجرائيات، بما في ذلك استخدام الرسائل والاستدعاء البعيد للإجراءات واتصالات المجموعات. وسنرى في هذا الفصل كيف تتعاون الإجراءات وتنسق أعمالها وتزامن بينها.

أمثلة على الأعمال التي تحتاج إلى تنسيق:

- إدارة المقطع الحرج.
- تخصيص الموارد.
- تحقيق العلاقة الزمنية بين الأحداث.

في الأنظمة المركزية يتم الاعتماد على طرق مبنية على وجود ذاكرة مشتركة مثل السيمافور والمراقب **monitor**، ولكن هذه الطرق لا تنجح في النظم الموزعة بسبب عدم وجود ذاكرة مشتركة.

النقاط التي ستطرح في هذا الفصل

- الزمن وقياسه، نظراً لأهميته بالنسبة للتزامن.
- خوارزميات الاستبعاد المتبادل **mutual exclusion**.
- التزامن عالي المستوى: المناقلات **transactions**.
- لمحة عن الاستعصاء **deadlock** في الأنظمة الموزعة.

2. مزامنة الميقاتية clock synchronization

من الطبيعي أن تكون عملية المزامنة في النظم الموزعة أكثر تعقيداً منها في الأنظمة المركزية، وذلك لأنها تحتاج إلى تطبيق خوارزميات موزعة، إذ لا يمكن في هذه الأنظمة تجميع المعلومات في مكان واحد واتخاذ القرار كما يحصل في الأنظمة المركزية.

خصائص عامة للخوارزميات الموزعة:

1. المعلومات اللازمة مبعثرة على عدة حواسيب.
2. لا يمكن للإجرائيات اتخاذ القرار إلا بناء على معلومات محلية.
3. يجب تلافي وجود نقطة مسببة للفشل **single point of failure**. من المفروض أن تكون المنظومات الموزعة أكثر وثوقية من المنظومات المركزية، بحيث تتابع الحواسيب العمل في حال توقف أحدها، ووجود نقطة مسببة للفشل هو غير مقبول البتة.
4. لا توجد ميقاتية مشتركة أو مصدر دقيق لمعلومة الزمن يعطي نفس الإجابة لجميع الإجرائيات في النظام.

ينتج من النقاط الثلاث الأولى أنه من غير الممكن أن تجمع كافة المعلومات في مكان واحد للمعالجة. فعلى سبيل المثال من غير المقبول إرسال كافة طلبات تخصيص الموارد إلى إجرائية مركزية تقوم بترتيب الطلبات والموافقة عليها أو تأجيلها بناء على معلومات مخزنة لديها. إذ سينتج عن ذلك ضغط كبير على تلك الإجرائية، كما يمكن أن تتحول بسهولة إلى نقطة مسببة للفشل مما يحد من وثوقية النظام.

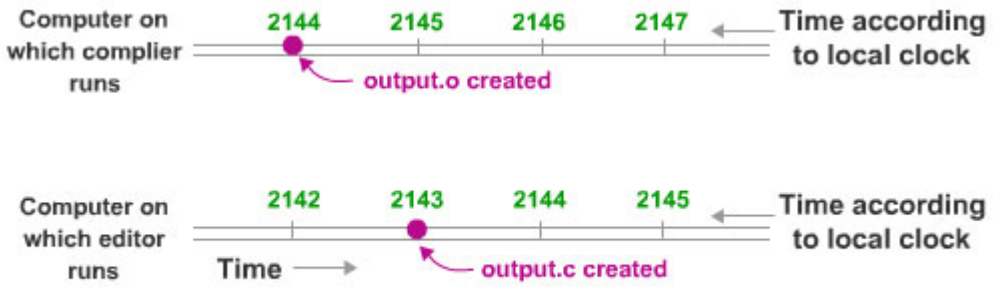
النقطة الأخيرة فائقة الأهمية، ففي النظام المركزي لا يوجد غموض في قيمة الزمن. عندما ترغب الإجرائية في معرفة الزمن، تقوم بإرسال طلب لنظام التشغيل وتحصل على إجابة النواة. وعندما تسأل الإجرائية **A** عن الزمن، وتتبعها إجرائية **B**، فمن المؤكد أن الإجابة التي ستحصل عليها **A** هي أقل من الإجابة التي ستحصل عليها **B**. ولكن هذه الخاصة ليست سهلة التحقيق في نظام موزع.

مثال (الشكل 1): تهدف تعليمة **Make** في نظام **UNIX** إلى الحد من عمليات ترجمة الملفات في حال البرامج الكبيرة التي تحتوي على عدد كبير من ملفات مصادر البرامج. وتعتمد على مبدأ مقارنة تواريخ تعديل الملفات بحيث لا تعاد ترجمة ملف مصدر إلا إذا كان تاريخ تعديل هذا الملف أكبر من تاريخ الملف التنفيذي (هذا شرح تبسيطي ولا داعي للدخول في تفاصيل عمل

التعليمة).

لنفترض أن تاريخ تعديل الملف **input.c** هو **2151** وأن تاريخ تعديل الملف **input.o** هو **2150**، سيستنتج برنامج **make** أن الملف **input.c** قد تغير وبالتالي سيقوم بإعادة ترجمة الملف، أما إذا كان تاريخ تعديل **input.c** هو **2144** فلن تحصل إعادة ترجمة.

في حالة نظام موزع لا يوجد اتفاق على قيمة الزمن، وبالتالي يمكن أن يتم تخزين نتيجة الترجمة على حاسوب أول، ويكون تاريخ تعديل الملف **output.o** هو **2144** ويجري تعديل الملف **output.c** المخزن على حاسوب آخر، ولكن بسبب بطئ الميقاتية الموجودة على ذلك الحاسوب تكون قيمة تاريخ التعديل هي **2143**. وبالتالي لن تتم عملية إعادة الترجمة ولن يتضمن الملف التنفيذي النهائي التعديلات التي قام بها المبرمج.



الشكل (1) مثال على مشكلة الاتفاق على الوقت في نظام موزع

السؤال المطروح إذا هنا هو التالي:

هل من الممكن توحيد أو مزامنة الميقاتيات المختلفة لكافة حواسيب النظام الموزع؟

الميقاتية المنطقية logical clock

يعتمد كل حاسوب في حساب الوقت على دارة متضمنة داخله تولد مقاطعة دورية تسمح بحساب الوقت. ويمكن أن يحصل انزياح في قيمة الزمن بسبب خطأ في إدخال الزمن لدى إقلاع الحاسوب أو بسبب استحالة كون المقاطعة دقيقة **100%** مهما كانت الدارة جيدة. ولكن لا توجد هناك مشكلة في ذلك عندما تكون الإجرائيات التي تقرأ الزمن تعمل على نفس الحاسوب. لأن الانزياح سيكون نفسه، وبالتالي فإن المقارنة تبقى صحيحة.

يتغير الوضع جذرياً عندما تكون هناك عدة وحدات معالجة تمتلك كل منها ميقاتية خاصة بها. فمن المستحيل التأكد أن كافة الدارات تعمل بنفس التواتر وبدون أي فرق، وأن أي انزياح سيحصل في أحدها سيكون نفسه في باقي الحواسيب. وبالتالي فإن الميقاتيات المختلفة ستعطي

قيماً مختلفة للوقت، ويعرف هذا الفرق باسم انحراف الميقاتية. ويمكن أن يؤدي هذا الانحراف إلى حصول خطأ في عمل العديد من البرمجيات التي تتوقع أن يكون الزمن مستقلاً عن الحاسوب الذي يتم عليه حساب الزمن (مثل برنامج **make** الذي رأيناه قبل قليل).

نعود إلى سؤالنا الأساسي: هل من الممكن أن تتم مزامنة هذه الميقاتيات للوصول إلى ميقاتية واحدة تعطي جواباً ليس فيه غموض؟

أورد لامبورت **Lamport** حلاً لهذه المشكلة في ورقة بحثية عام **1978**، أظهر فيها أنه قد يكون ذلك ممكن، مبيناً أنه ليس من الضرورة أن يكون تزامن الميقاتيات مطلقاً. فإذا لم يكن هناك تخاطب بين إجرائيتين فلا داعي لمزامنة الميقاتيات المرتبطة بكل منهما. كما بين أنه ليس من المهم أن تتفق الإجرائيات على قيمة الوقت، وإنما على ترتيب الأحداث. فإذا أخذنا المثال السابق المتعلق ببرنامج **make**، ليس من الضروري أن يكون هناك اتفاق على تواريخ تعديل الملفات وإنما على العلاقة بين التواريخ (الملف **input.c** هو أحدث أو أقدم من الملف **input.o**).

في كثير من الحالات يكفي أن تتفق الإجرائيات بين بعضها البعض على قيمة الميقاتية، ولو كانت هذه القيمة مختلفة عن الزمن الفعلي. وتعرف الميقاتية في هذه الحالة باسم الميقاتية المنطقية **logical clock**. عندما يضاف إلى تلك الضوابط ما يفرض عليها بألا يزيد انحرافها عن الزمن الفعلي بنسبة ما، فإن الميقاتية تعرف هنا باسم الميقاتية الفيزيائية.

نقوم بتعريف علاقة حدث-قبل **happensbefore**:

a ~ b يعني أن **a** حصل قبل **b**، وهذا يعني أن كافة الإجرائيات ضمن المنظومة ترى أن الحدث **b** قد تم بعد الحدث **a**. ويمكن ملاحظة هذه العلاقة في حالتين:

1. إذا كان **a** و **b** حدثين من نفس الإجرائية، تكون لحظة حدوثهما مقاسة بنفس الميقاتية وبالتالي فإن المرجعية الزمنية واحدة. وفي هذه الحالة تكون العلاقة **a ~ b** مكافئة لكون زمن وقوع **a** أصغر من زمن وقوع **b**.
2. إذا كان **a** هو حدث إرسال رسالة من إجرائية أولى، و **b** هو حدث استقبال الرسالة لدى إجرائية ثانية، فإن العلاقة **a ~ b** هي أيضاً محققة، إذ لا يمكن أن تستلم رسالة قبل إرسالها، أو حتى في نفس وقت إرسالها، إذ تحتاج إلى إلى زمن أكبر من الصفر للوصول إلى وجهتها.

ملاحظات:

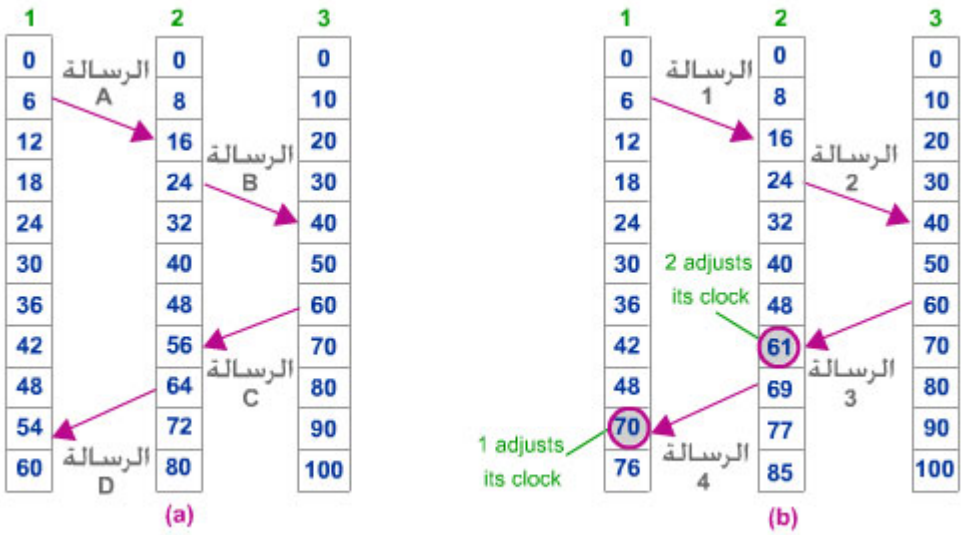
- من الواضح أن العلاقة هي متعدية، فإذا كان **a ~ b** و **b ~ c** فإن **a ~ c**.

- يقال أن الحدثين **a** و **b** متواردان **concurrent** في حال عدم وجود أية علاقة بينهما (**a ~ b** غير صحيحة و **b ~ a** غير صحيحة)، أي أنه لا يمكن الجزم بأي منهما وقع قبل الآخر.

مانحتاجه هو إمكانية تعريف تابع يعطي لكل حدث **a** قيمة **C(a)** تسمى بلحظة وقوع الحدث، وتتفق عليها كافة الإجرائيات. ويجب أن تحقق هذه القيم الشرط: **a ~ b => C(a) < C (b)**.

وبحسب تعريف العلاقة، إذا كان الحدث **a** هو إرسال الرسالة، والحدث **b** هو استقبالها، فإن اللحظة المرتبطة بـ **a** يجب أن تكون أصغر من تلك المرتبطة بـ **b**. كما يجب أن يكون تابع الزمن متزايداً دوماً ولا عودة فيه إلى الوراء.

يعرض الشكل **2-a** ثلاث إجرائيات تعمل على حواسيب مختلفة ولكل منها ميقاتيته الخاصة به. وبعد مرور ستة دقات للميقاتية في الإجرائية **0**، نرى أن هناك **8** دقات في الإجرائية **1** و**10** دقات في الإجرائية **2**، إذ أن كل ميقاتية تعمل بتواتر مختلف.



الشكل (2) مثال على خوارزمية لامبورت

في اللحظة **6**، تقوم الإجرائية **0** بإرسال الرسالة **A** إلى الإجرائية **1**. وسيكون قياس الزمن الذي استغرقته الرسالة للوصول إلى وجهتها مربكاً.

- تحتوي الرسالة على لحظة الإرسال: **6**.
- تصل الرسالة إلى وجهتها في اللحظة **16** بحسب الإجرائية المستقبلة.
- استنتاج: **10** دقات لوصول الرسالة **A**، وبنفس الطريقة تحتاج الإجرائية **2** لإرسال الرسالة **B** إلى الإجرائية **3** إلى **12** دقة، وهما قيمتان ممكنتان.
- أما الرسالة **C**، فسترسل في اللحظة **60** وتصل في اللحظة **56**، والرسالة **D** ترسل في

اللحظة **64** وتصل في اللحظة **54**، وهي قيم ببساطة مستحيلة لأنها تعني أن الزمن يعود إلى الوراء !!

نتيجة: لا يمكن الاعتماد على الميقاتية المحلية لكل إجرائية في تحديد الزمن.

تعتمد خوارزمية لامبورت على علاقة **happened-before**، فالرسالة **C** غادرت في اللحظة **60**، ويجب أن تصل في اللحظة **61** أو بعدها على أقل تقدير. تطبق الخوارزمية الخطوات التالية:

- تحتوي كل رسالة على زمن الإرسال بحسب توقيت المرسل.
- عندما تصل الرسالة إلى وجهتها، وفي حال كان التوقيت لدى المستقبل أقل من توقيت الإرسال المتضمن في الرسالة، تعدل قيمة الميقاتية لدى المستقبل لتصبح مساوية إلى توقيت الإرسال **+ 1**.
- عندما تقوم الإجرائية بإرسال رسالة تقوم بزيادة الميقاتية الموجودة لديها بمقدار **1**.

من السهل التأكد من كون العلاقة التالية محققة:

$$a \sim b \Rightarrow C(a) < C(b)$$

نلاحظ أن العكس ليس صحيحاً، أي أنه من غير الممكن استنتاج علاقة ترتيب بين حدثين انطلاقاً من الميقاتية المنطقية لوحدها. ويمكن الاستفادة من الميقاتية المنطقية في تعريف علاقة ترتيب كلية في حال استخدام رقم الإجرائية التي يحصل فيها الحدث لمعالجة الحالة التي تتساوى فيها قيمة الميقاتية بين الإجرائين: ففي حال حدوث حدثين ضمن الإجرائية **1** و **2** يقترن كل منهما باللحظة الافتراضية **50** فإن نتيجة المقارنة ستكون **50.2 > 50.1**.

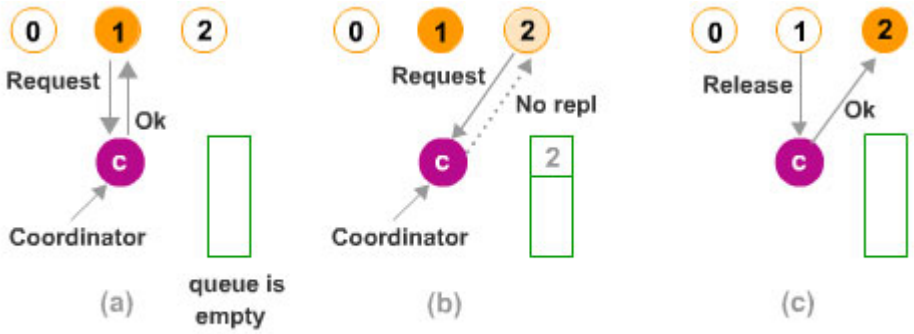
3. الاستبعاد المتبادل

تعتبر مسألة الاستبعاد المتبادل من أهم مسائل التنسيق والتزامن بين الإجرائين. وغالباً ما يتم تحقيقها عبر تعريف ما يسمى بالمقاطع الحرجة، وهي مجموعة تعليمات لا يجوز أن تتواجد ضمنها أكثر من إجرائية واحدة في لحظة ما. وفي المنظومات وحيدة المعالجات يتم تحقيق المقاطع الحرجة باستخدام السيمافور والمراقب **monitor** وآليات مشابهة. سنحاول في هذا المقطع الاطلاع على خوارزميات مختلفة لحل المعضلة.

خوارزمية مركزية

وهي الطريقة الأكثر سهولة، وتعمل على الشكل التالي:

1. يتم انتخاب إجرائية تنسيق **coordinator**.
2. عندما تريد إجرائية دخول المقطع الحرج، ترسل طلبها إلى إجرائية التنسيق وتذكر ضمن الطلب اسم المقطع الذي ترغب في دخوله، وتنتظر الإجابة على طلبها.
3. في حال عدم وجود إجرائية أخرى داخل المقطع الحرج، ترسل إجرائية التنسيق جواباً بالموافقة، وعندما تصل الإجابة إلى الإجرائية الطالبة تدخل المقطع الحرج.
4. في حال وجود إجرائية أخرى، تؤخر إجرائية التنسيق الجواب حتى لايبقى هناك أية إجرائية داخل المقطع الحرج.



الشكل (3) الخوارزمية المركزية لإدارة المقطع الحرج

من السهل التحقق من أن الخوارزمية تحقق الاستبعاد المتبادل، وأنها عادلة لأنها تجيب الطلبات بحسب ترتيب ورودها وبالتالي لا توجد مجاعة (انتظار الإجرائية إلى مالانهاية). وهي أيضاً سهلة التحقيق ولا تحتاج إلى أكثر من ثلاث رسائل لكل طلب، كما يمكن تعميمها لإدارة موارد أخرى وليس فقط المقاطع الحرجة.

أما المساوئ فتتلخص في النقاط التالية:

- تحويل إجرائية التنسيق إلى نقطة فشل، فإذا توقفت تفشل المنظومة كلها.
- لا يمكن للإجرائية الطالبة التمييز بين توقف إجرائية التنسيق وبين انتظار طويل بسبب وجود عدد كبير من الإجرائيات قبلها.
- يمكن أن تؤدي المركزية إلى حصول اختناق في حال كون المنظومة كبيرة.

خوارزمية موزعة

كثيراً ما تكون مشكلة نقطة الفشل غير مقبولة البتة، مما يفرض البحث عن خوارزميات موزعة بهدف تلافي هذه النقطة الواحدة.

من أهم الأمثلة خوارزمية ريكار وأغراوالا **Ricart & Agrawala**، والتي تعتمد على فرض ترتيب كلي لكافة الأحداث في النظام، بحيث يمكن تحديد أسبقية حدث بالنسبة لآخر بدون أي غموض أو التباس. ويمكن اعتماد خوارزمية لامبورت كإحدى الطرق الممكنة لتعريف هذه العلاقة.

تعمل الخوارزمية بحسب الخطوات التالية:

- عندما تريد إجرائية الدخول في مقطع حرج، تنشئ رسالة فيها اسم المقطع الحرج الذي تريد الدخول فيه، بالإضافة إلى رقم الإجرائية ولحظة إنشاء الرسالة.
- ترسل الإجرائية الرسالة إلى كافة الإجرائيات الأخرى، بما فيها نفسها.
- يجب أن يكون إرسال الرسالة موثوقاً، أي أن الرسالة ستصل حتماً إلى جميع الإجرائيات وستستلم الإجرائية المرسلة وصل استلام بذلك.
- بعد إرسال الطلبات، تنتظر الإجرائية وصول موافقة كافة الإجرائيات الأخرى قبل دخولها في المقطع الحرج، وتدخل المقطع فور اكتمال الموافقات.
- عند ورود طلب موافقة إلى إجرائية من إجرائية أخرى، تتوقف إجابتها على وضع الإجرائية المستقبلة بالنسبة للمقطع الحرج المذكور في الرسالة. ويمكن تمييز الحالات التالية:

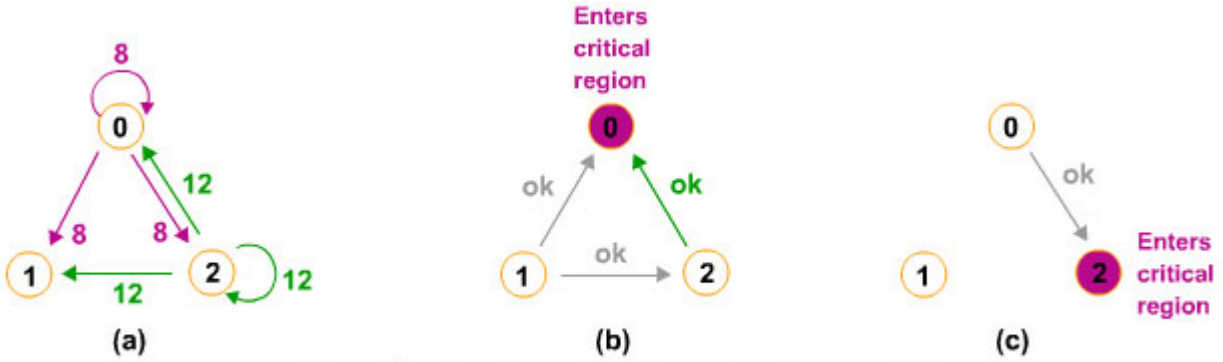
1. في حال كانت الإجرائية خارج المقطع الحرج ولا تريد الدخول إليه ترسل الموافقة إلى الإجرائية الطالبة.
2. في حال كانت الإجرائية المستقبلة داخل المقطع الحرج، لا تجيب على

الطلب وتضع الإجرائية الطالبة ضمن رتل انتظار.

3. في حال كانت الإجرائية قد أرسلت طلباً لدخول المقطع الحرج، ولكنها لم تدخل بعد، فإنها تقارن بين تاريخ الرسالة الواردة وتاريخ الرسالة التي أرسلتها. وتكون الأولوية للرسالة ذات التاريخ الأقل: إذا كان تاريخ الرسالة الواردة أقل، ترسل الإجرائية المستقبلة رسالة موافقة، وإلا فإنها تضع الرسالة الواردة في رتل الانتظار ولا ترسل أية إجابة.

- عند خروج الإجرائية من المقطع الحرج ترسل الموافقة لكافة الإجرائيات المنتظرة في رتل الانتظار لديها، وتحذفها من الرتل.

يظهر الشكل 4 مثالاً عن عمل الخوارزمية، من الواضح أن الخوارزمية تعمل بنجاح في حال عدم وجود تضارب. يظهر الشكل 4-a حالة وجود إجرائيتين ترغبان في دخول المقطع الحرج في نفس اللحظة.



الشكل (4) مثال على عمل خوارزمية ريكار وأغراوالا

- ترسل الإجرائية رقم 0 طلباً إلى كافة الإجرائيات وتكون قيمة اللحظة المرتبطة به هي 8، وتقوم الإجرائية رقم 2 في الوقت نفسه بإرسال طلب إلى كافة الإجرئيات ولكن مع لحظة زمنية مرتبطة هي 12. أما الإجرائية الأولى فهي غير مهتمة بالدخول إلى المقطع الحرج، وبالتالي فسترسل OK إلى الإجرائيتين: 0 و 2.
- سترى الإجرائية 2 أنها قد خسرت المقارنة لصالح 0، وبالتالي ترسل لها رسالة OK.
- الإجرائية 0 ستحصل على نفس النتيجة، وبالتالي تضع الإجرائية 2 ضمن الرتل لديها، وستستلم رسالة OK من جميع الإجرائيات وعندها تدخل المقطع الحرج كما يظهر الشكل 4(b).

- عندما تنتهي الإجرائية **0** من المقطع الحرج ترسل رسالة **OK** إلى الإجرائية **2** لتعلمها بأنه بات بإمكانها الدخول إلى المقطع الحرج، وذلك كما يظهر الشكل **4(c)**.
- سبب صحة عمل الخوارزمية هو عدم وجود أي غموض في علاقة الترتيب وبالتالي فإن الإجرائية **2** تتفق مع الإجرائية **0** في تقييمها لترتيب الرسائل.

تقييم الخوارزمية

ليس من الصعب إثبات أن الخوارزمية تعمل بدون استعصاء أو مجاعة. عدد الرسائل اللازمة لكل طلب دخول هو **2(n-1)** حيث **n** هو عدد الإجرائيات في النظام.

لا توجد نقطة فشل وحيدة كما هو الحال في الخوارزمية الموزعة، ولكن للأسف هناك **n** نقطة فشل. إذ أن توقف أية إجرائية عن العمل فلن ترسل موافقتها إلى الإجرائية الطالبة وبالتالي لن يكون بإمكان تلك الإجرائية أن تدخل المقطع الحرج. وباعتبار أن احتمال تعطل واحدة من الإجرائيات هو **n** مرة أكبر من احتمال تعطل إجرائية واحدة، فإننا نكون هنا قد استبدلنا خوارزمية سيئة بخوارزمية أسوأ منها بـ **n** مرة وتحتاج إلى عدد أكبر بكثير من الرسائل.

يمكن التغلب على هذه المشكلة عبر إرسالة رسالة رفض في حال عدم الموافقة، وتقوم الإجرائية الطالبة التي تستقبل رسالة رفض بإعادة الطلب. وتنتظر لزمن محدود قبل إعادة الطلب في حال عدم ورود جواب، مما يسمح لها باكتشاف وجود العطل. ولكننا هنا سنحتاج إلى عدد رسائل أكبر بكثير من العدد الأصلي **2(n-1)**.

هناك أيضاً مشكلة تكمن في معرفة قائمة الإجرائيات الموجودة ضمن النظام بحيث تضمن الإجرائية المرسلة وصول الرسائل إلى كافة الإجرائيات. هذه المسألة تعرف باسم خوارزميات اتصال المجموعات **group communication algorithms**، وسنتعرض لها لاحقاً في هذا الفصل. وهذه الخوارزميات تحتاج إلى حل مسألة انضمام الإجرائية إلى المجموعة وتركها سواء طواعية أو نتيجة خطأ. وتعمل هذه الخوارزمية بشكل أمثلي مع المجموعات الصغيرة من الإجرائيات التي لا تتغير عضويتها.

في النهاية، كانت إحدى مشكلات الخوارزمية الموزعة هي معالجة إجرائية واحدة لكل الطلبات. في هذه الخوارزمية ستكون كافة الإجرائيات في نفس الوضع إذ ستكون مضطرة لمعالجة كافة الطلبات. وبالتالي لا يتوقع أن يكون أداء الخوارزمية أفضل منه في حال الخوارزمية الموزعة.

يمكن تحسين الأداء بعض الشيء إذا لاحظنا أنه ليس من الضروري الحصول على موافقة كافة

الإجرائيات، إذ يمكن الاكتفاء بموافقة أغلبية الإجرائيات (النصف + 1). طبعاً هذا يعني أن إجرائية أرسلت موافقة لإجرائية ما على دخول المقطع الحرج لن ترسل موافقة لإجرائية أخرى قبل خروج الأولى منه.

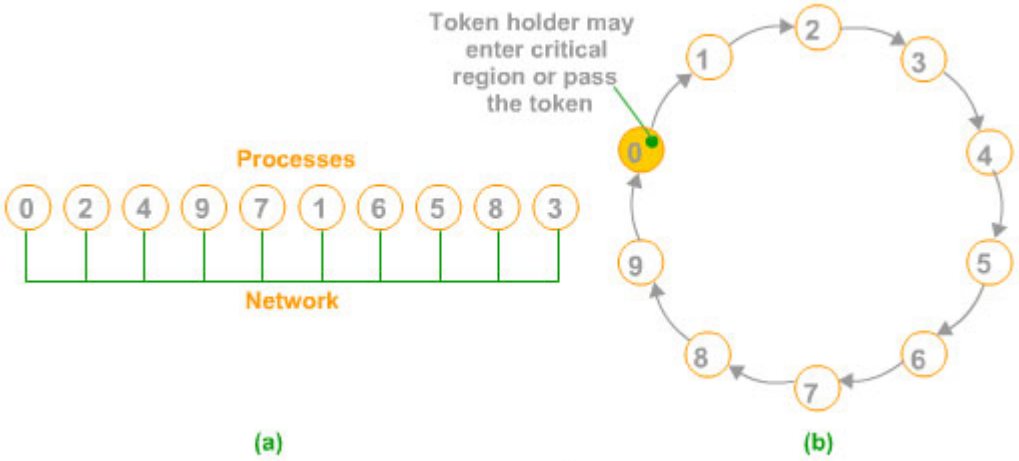
بالنتيجة: هذه الخوارزمية أسوأ بكثير من الخوارزمية المركزية: فهي أبطأ أعقد وأكثر كلفة وأقل وثوقية. والسؤال: لماذا ندرسها إذاً، الجواب بسيط: للتأكيد على إمكانية وجود خوارزمية موزعة. وهذا بحد ذاته أمر ضروري في مقرر اسمه "النظم الموزعة" !!

خوارزمية حلقية

الخوارزمية التالية مختلفة كلياً عن سابقتها. نستخدم هنا شبكة من نوع الناقل **bus**، ويقصد بذلك أنها ترسل رسالة واحدة في لحظة ما. يتم تعريف ترتيب بين الإجرائيات على شكل حلقة، وبحيث يكون لكل إجرائية مكان واضح ضمن الحلقة. يمكن مثلاً أن يتم الترتيب بحسب العنوان الشبكي لكل إجرائية ولكن هذا ليس مهما، فالمهم هو أن تعرف كل إجرائية من هي الإجرائية التي تليها في الترتيب.

عندما يتم تأسيس الحلقة، تعطى الإجرائية رقم **0** شارة **token**، وتدور الشارة ضمن الحلقة عن طريق إرسال رسالة من نوع **point-to-point** من الإجرائية إلى الإجرائية التالية. وعندما تحصل الإجرائية على الشارة يكون من حقها الدخول إلى المقطع الحرج، فإذا كانت ترغب بذلك تدخل إلى المقطع وتعيد تمرير الشارة بعد الخروج من المقطع الحرج. وفي حال عدم الرغبة تكتفي بتمرير الشارة إلى الإجرائية التالية. بالنتيجة، عندما لا يكون هناك من يريد دخول المقطع الحرج فإن الشارة تدور بسرعة عالية ضمن الحلقة.

من السهل التحقق من صحة الخوارزمية، فهي بحسب تصميمها لا تسمح للإجرائية بدخول المقطع الحرج ما لم تكن حاملة للشارة، وهناك شارة واحدة في الحلقة، وبالتالي لايمكن أن تكون هناك أكثر من إجرائية داخل المقطع الحرج. لا يمكن أيضاً أن تحصل مجاعة لأن الشارة تدور بالترتيب، فزمن الانتظار الأقصى لأية إجرائية تريد الدخول إلى المقطع الحرج هو **n-1** * الزمن الأقصى الذي يمكن أن تقضيه إجرائية داخل المقطع الحرج.



الشكل (5) مبدأ الخوارزمية الحلقية

طبعاً لهذه الخوارزمية مشاكلها وسيئاتها. والسيئة الأهم هي ضياع الشارة، ففي هذه الحالة يجب إعادة توليد الشارة، والمشكلة هي في اكتشاف حالة الضياع. لأنه لا يمكن التمييز بين حالة استخدام طويل للشارة ناتج عن دخول في مقاطع حرجة لفترة طويلة وبين حالة ضياع الشارة.

هناك أيضاً مشكلة توقف إحدى الإجرائيات، ولكن يمكن حلها بطريقة أسهل من الحالات الأخرى، إذ يكفي أن تكون عملية تسليم الشارة تحتاج إلى رسالة تأكيد من الإجرائية المستلمة. وفي هذه الحالة ستكتشف الإجرائية السابقة للإجرائية المتوقفة ذلك التوقف عند محاولتها إعطاءها الشارة، إذ ستفشل العملية. ويمكن بعدها إعادة تشكيل الحلقة من الإجرائيات المتبقية.

مقارنة الخوارزميات الثلاث

إذا قارننا الخوارزميات الثلاث التي أوردناها في هذا المقطع فسنحصل على الجدول التالي الذي يعتمد على ثلاث نقاط أساسية: عدد الرسائل اللازمة لكل إجرائية من أجل دخول ومغادرة المقطع الحرج، والفترة اللازمة قبل أن تتمكن الإجرائية من الدخول، وما هي أهم المشكلات التي تعيق عمل الخوارزمية.

الخوارزمية	عدد الرسائل لكل دخول ومغادرة	زمن الانتظار قبل الدخول (الواحدة هي زمن إرسال الرسالة)	أهم المشكلات
مركزية	3		توقف المنسق
موزعة	2(n-1)	2(n-1)	توقف أية إجرائية
حلقية	1 إلى مالانهاية	0 إلى n-1	ضياع الشارة (Token)

إن الخوارزمية المركزية هي الأبسط والأكثر فاعلية، فهي تتطلب فقط ثلاث رسائل لدخول ومغادرة المقطع الحرج. مقابل n-1 رسالة طلب يضاف إليها n-1 رسالة موافقة للخوارزمية الموزعة. أما في حالة الخوارزمية الحلقية فالعدد متغير: فعندما تكون كل إجرائية ترغب باستمرار في دخول المقطع الحرج (حالة حدية) فسينتج عن كل مرور للشارة دخول، وبالتالي ما يلزم في هذه الحالة هو رسالة واحدة. أما إذا لم تكن هناك رغبة في الدخول فستدور الشارة عدداً غير محدود من الدورات وبالتالي عدد الرسائل غير محدد.

أما عن الزمن اللازم لدخول الإجرائية اعتباراً من اللحظة التي تعبر فيها عن رغبتها في الدخول فهو ايضاً متغير بين الخوارزميات الثلاث، فهو رسالتين في حالة الخوارزمية المركزية، و2(n-1) رسالة في حالة الخوارزمية العادية، أما في حالة الخوارزمية الحلقية فيتراوح بين 0 (وصلت الشارة لتوها) و n-1 (تم الطلب بعد مغادرة الشارة).

وتعاني الخوارزميات الثلاث من ضعف حاد في جانب الوثوقية، إذ يمكن أن يؤدي توقف إجرائية إلى فشل عمل المنظومة بأكملها. وهذه من أكبر العيوب والتي لا يمكن القبول بها إلا في حال كون المنظومة صغيرة وعدد الأعطال محدوداً.

4. المناقلات

تندرج كافة آليات التزامن التي درسناها حتى الآن ضمن إطار التزامن منخفض المستوى (كالسيمافور)، وهي تتطلب من مطور البرمجيات الانتباه بعناية فائقة إلى كافة النقاط المتعلقة بالاستبعاد المتبادل. والمطلوب هو مستوى أعلى من التجريد يريح المطور من تلك التفاصيل، ويسمح له بالتركيز أكثر على عمل الخوارزميات. وتعرف هذه الآلية في الأنظمة الموزعة باسم المناقلات **transactions**.

والمبدأ الأساسي في عمل المناقلات هو التالي: تقوم إحدى الإجرائيات بمباشرة عمل ما بالاتفاق مع باقي الإجرائيات، ويتبع ذلك إنشاء وحذف وتغيير العديد من الأغراض ضمن النظام، وفي النهاية تأتي لحظة الحسم: تطلب الإجرائية التي أطلقت العملية بالطلب من الجميع بالموافقة على ما تم حتى لحظتها، وفي حال الموافقة تصبح النتائج دائمة، وإلا يعاد النظام إلى نفس الحالة التي كان عليها قبل إطلاق المناقلة. بمعنى آخر فإن المناقلة عملية ذرية **atomic** تعمل بمبدأ كل شيء أو لاشيء، وهذا ما يسهل كثيراً من عمل مطور التطبيقات الذي لم يعد عليه أن يسأل نفسه في كل تعليمة "ماذا سيحصل في حال وقوع خطأ هنا؟"

دعونا نأخذ مثالاً بسيطاً عن تطبيق مصرفي يقوم بإدارة قاعدة بيانات حسابات المشتركين. يتصل زبون عن طريق الإنترنت ليقوم بتنفيذ عملية تحويل من حسابه إلى حساب آخر. ويقوم التطبيق المصرفي الخاص المثبت على حاسوب الزبون بطلب تنفيذ العملية على مرحلتين:

1. سحب المبلغ من الحساب الأول.
2. إضافة المبلغ إلى الحساب الثاني.

في حال توقف الاتصال بعد السحب وقبل الإضافة تتبخر الأموال في الهواء (طبعاً في حال عدم وجود أية احتياطات)!!

أما في حالة استخدام المناقلات فلن تكون هناك مشكلة ضياع، إذ ستعود المنظومة إلى ماكانت عليه قبل عملية السحب الأولى.

نموذج المناقلات

نعرف في هذا المقطع المحددات الأساسية لنموذج المناقلات. تتكون المنظومة من عدد من الإجرائيات التي تتصل بين بعضها البعض باستخدام رسائل غير موثوقة (أي أنها يمكن أن تضيع). ولكن نفترض وجود طبقة اتصالات مسؤولة عن معالجة الضياع الناتج عن الشبكة.

التخزين المستقر:

تفترض المناقلات وجود تخزين مستقر، وهو مصمم بحيث يقاوم كافة الأعطال باستثناء الأحداث شديدة القوة كالزلازل والفيضانات. يمكن تحقيق التخزين المستقر باستخدام زوج من الأقراص الصلبة العادية التي تعمل بمبدأ المرآة: القرص الثاني هو صورة مطابقة للقرص الأول.

في حال حصول عطل للنظام بعد تعديل القرص الأول وقبل تعديل الثاني كما يظهر في الجزء b، تتم مقارنة القرصين مقطعاً مقطعاً، وعند حصول اختلاف يفترض أن القرص الأول هو الصحيح وتنسخ القيمة على القرص الثاني. وعند الانتهاء من العملية يكون القرصان متطابقين.

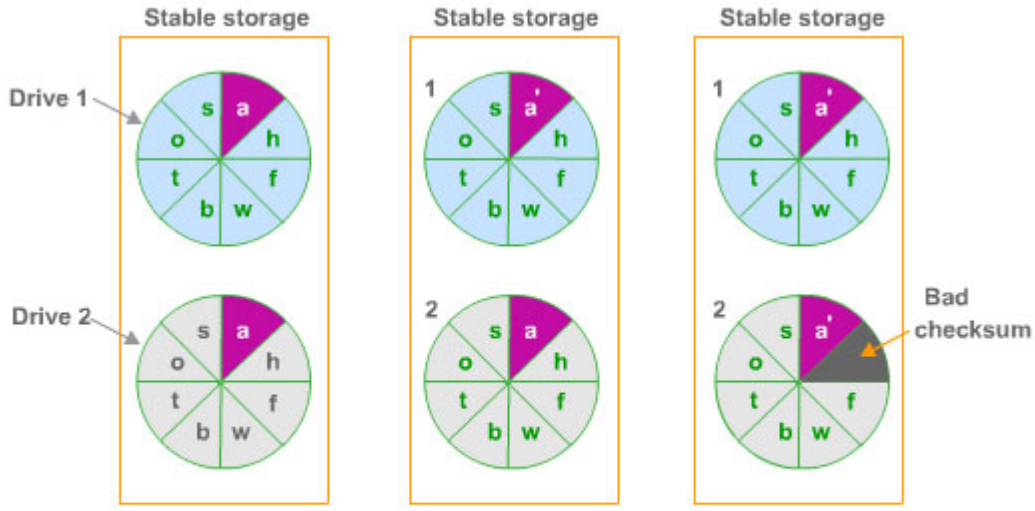


Fig. 3-15. Stable storage Crash after drive 1 is updated Bad spot.

الشكل (6) التخزين المستقر باستخدام قرصين

يمكن أيضاً لهذه الآلية معالجة الأعطال الناتجة عن تقادم القرص، ففي لحظة ما يمكن أن يظهر خطأ على أحد المقاطع، وفي هذه الحالة تنسخ القيمة من القرص الآخر.

استدعاءات المناقلات:

تحتاج عملية برمجة المناقلات إلى عدد من الاستدعاءات التي يمكن أن يقدمها نظام التشغيل أو بيئة التنفيذ. ومثال عليها:

a. **BEGIN_TRANSACTION**: لبدء المناقلة.
b. **END_TRANSACTION**: انتهاء المناقلة بنجاح وحفظ كافة التعديلات.
c. **ABORT_TRANSACTION**: إنهاء المناقلة بالفشل والعودة إلى الحالة قبل بدء المناقلة.
d. **READ**: قراءة بيانات من ملف أو أي غرض آخر.
e. **WRITE**: كتابة بيانات إلى ملف أو أي غرض آخر.

تعتمد القائمة الدقيقة لكافة الاستدعاءات على نوع الأغراض المستخدمة في المناقلة، ففي منظومة بريد إلكتروني سيكون هناك استدعاءات لإرسال واستقبال ومتابعة رسالة، أما في منظومة مصرفية فستكون الاستدعاءات مختلفة حتماً.

خصائص المناقلات:

للمناقلات أربع خصائص أساسية، (يتم ترميز هذه الخصائص باستخدام الأحرف الأولى **ACID**):

1. ذرية **Atmoic**: ويقصد بها أن المناقلة تنفذ بشكل غير قابل للتجزئة: إما بأكملها أو بدون تنفيذ. ولا تظهر القيمة الانتقالية للإجرائيات غير المعنية بالمناقلة، فقط القيمة النهائية في حال تثبيتها أو تبقى البدائية في حال الفشل.
2. متسقة **Consistent**: لا تخرق المناقلة الخصائص الأساسية المثبتة للنظام. فعلى سبيل المثال، الخاصة الأساسية للنظام هي ثبات المجموع الكلي للكتلة النقدية، ويجب أن يكون مجموع الأرصدة بعد أي عملية تحويل داخلية هو نفس ما كان عليه قبل عملية التحويل.
3. معزولة **Isolated**: لا تتداخل المناقلات ببعضها البعض. ويقصد بذلك أن نتيجة تنفيذ عدة مناقلات متواردة تكافئ نتيجة تنفيذ تسلسلي ما للمناقلات (يكفي أن يوجد تسلسل واحد لتكون هذه الخاصة صحيحة). ويظهر الشكل **7** أدناه ثلاث مناقلات متداخلة بطرق مختلفة، اثنتان منهما لها تنفيذ تسلسلي مكافئ وبالتالي هي مقبولة، والثالثة ليس لها تنفيذ تسلسلي مكافئ وبالتالي هي مرفوضة.
4. دائمة **Durable**: عندما يتم تثبيت القيم في نهاية المناقلة تكون هذه القيمة دائمة. ولا يمكن أي عطل بعد انتهاء المناقلة أن يؤثر على هذه القيم.

BEGIN_TRANSACTION
x=0;
x=x+1;
END_TRANSACTION
(a)
	BEGIN_TRANSACTION
	x=0;
	x=x+2;
	END_TRANSACTION
	(b)
		BEGIN_TRANSACTION
		x=0;
		x=x+3;
		END_TRANSACTION
		(c)

Time ⟶

Schedule1	x=0;	x=x+1;	x=O;	x=x+2;	x=O;	x=x+3;	Legal
Schedule2	x=0;	x=0;	x=x+1;	x=x+2;	x=0;	x=x+3;	Legal
Schedule3	x=0;	x=0;	x=x+1;	x=0;	x=x+2;	x=x+3;	Legal

(d)

الشكل (7) عزل المناقلات والتنفيذ التسلسلي المكافئ

تحقيق المناقلات

سنشرح باختصار كيف يتم تحقيق المناقلات في نظام موزع.

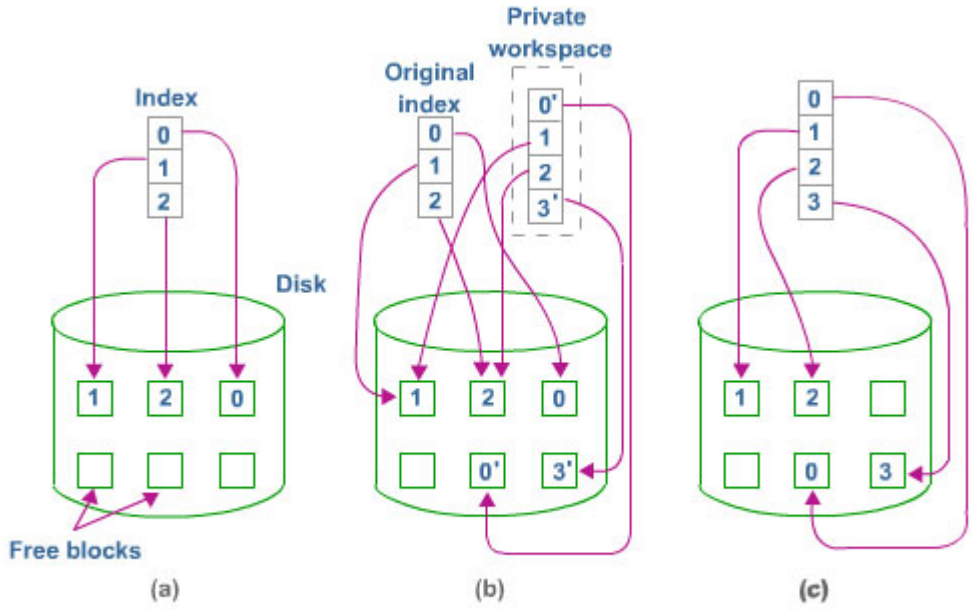
الطريقة الأولى:

وتعرف هذه الطريقة باسم فضاء العمل الخاص **private workspace**. فعندما تطلب إجرائية الدخول في مناقلة يتم تخصيص فضاء عمل خاص لها وتنسخ كافة الأغراض التي تستخدمها الإجرائية في هذا الفضاء، وأثناء عمل المناقلة تذهب كافة عمليات الكتابة والقراءة إلى ذلك الفضاء عوضاً عن الأغراض الحقيقية. وتنسخ الأغراض من فضاء العمل الخاص على الأغراض الحقيقي لدى نجاح المناقلة.

المشكلة هنا هي في وجود كلفة كبيرة لنسخ كافة الأغراض، وهناك عدة آليات للتسريع، منها عدم نسخ الأغراض التي تقرأ فقط وتتم القراءة من الغرض الأساسي مباشرة بدون نسخ.

ويكون فضاء العمل الخاص فارغاً في البداية ويتم نسخ الأغراض التي تعدل عند أول طلب للكتابة فيها، وفي حال الملفات هناك تسريع إضافي إذ لا يتم نسخ الملف بأكمله، وانما يتم نسخ دليل الملف **index** الذي يحتوي قائمة وحدات الملف **blocks**. وتنسخ كل وحدة من وحدات الملف عند تعديله ضمن فضاء العمل الخاص، وتعدل نسخة الدليل بحيث تشير إلى الوحدة المنسوخة عوضاً عن الوحدة الأصلية، وتعرف هذه الآلية باسم وحدات الظل **shadow blocks**. وهي معروضة في الشكل 8. وكما يظهر الشكل **8-b**، فإن الإجرائيات التي تنفذ المناقلة سترى الملف المعدل، أما باقي الإجرائيات فستظل ترى الملف الأساسي.

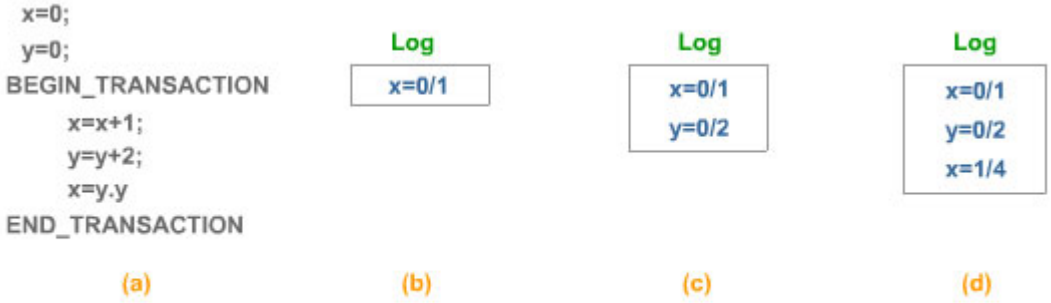
إذا فشلت المناقلة فسيتم حذف كافة الأغراض المحلية وإعادة الوحدات إلى قائمة الوحدات المتاحة. أما في حالة نجاح المناقلة، فسيتم تعديل الدليل الأصلي بحيث يشير إلى الوحدات الجديدة عوضاً عن الوحدات القديمة. وتعاد الوحدات التي أصبحت غير مستخدمة إلى قائمة الوحدات المتاحة.



الشكل (8) فضاء العمل الخاص باستخدام وحدات الظل

الطريقة الثانية:

الطريقة الثانية لتحقيق المناقلات هي طريقة السجل الاستباقي **writeahead log** وتعرف أحياناً باسم قائمة التوقعات. وبهذه الطريقة يمكن تعديل الملفات، ولكن قبل أن يتم تغيير أي وحدة، يجب إضافة سطر إلى سجل دائم مخزن على التخزين المستقر يتضمن المعلومات التالية: المناقلة التي تنفذ التغيير، الملف والكتلة التي تم تغييرها، والقيم الجديدة والقديمة. ويتم تنفيذ التغيير بعد القيام بحفظ المعلومات السابقة في السجل.



الشكل (9) مثال على عمل السجل الاستباقي

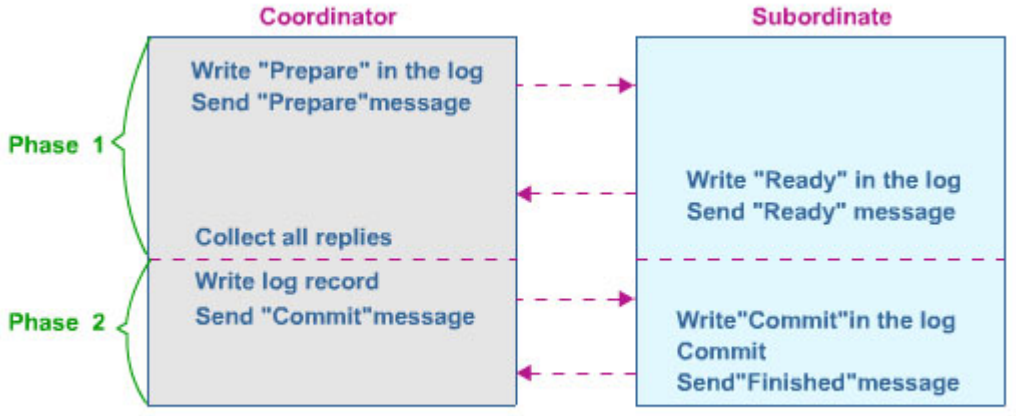
يظهر الشكل **9** آلية عمل السجل الاستباقي: لدينا مناقلة بسيطة تستخدم متحولين مؤسسين على قيمة **0**. يحتوي السجل على عدد من الأسطر تتضمن القيمة القديمة والجديدة للمتحولات يفصلها حرف "/". ويظهر الشكل تطور السجل مع تنفيذ كل تعليمة من التعليمات.

في حال نجاح المناقلة وتثبيت القيمة، يتم إضافة سطر تثبيت إلى السجل، ولا يتم أي تغيير على البيانات. أما في حال الفشل فيتم العودة عبر السجل سطراً سطراً إلى ما قبل بداية المناقلة، وتسمى هذه العملية بالتراجع **rollback**.

يفيد هذا السجل أيضاً في استرجاع العمل بعد التوقف، فإذا افترضنا أن الإجرائية التي تنفذ المناقلة توقفت بعد قيامها بكتابة السجل الذي يظهر في الشكل **9-d** ولكن قبل أن تقوم فعلياً بتغيير قيمة المتحول **x**، سيقوم نظام التشغيل بإعادة تنفيذ كافة التعليمات الموجودة في السجل والخاصة بالمناقلات غير المنتهية واحدة تلو الأخرى. ويتم تعديل قيمة المتحول **x** لتصبح مطابقة لما هي عليه في السجل، فإذا كانت **4** هذا يعني أن التوقف حصل بعد التعديل وبالتالي لاداعي للتعديل، أما إذا كانت **1** فهذا يعني أن التوقف حصل قبل التعديل وبالتالي يجب أن تعدل لتصبح **4**.

التثبيت على مرحلتين

تقتضي خاصة الذرية أن تجرى عملية التثبيت المقترنة بالانتهاء الناجح للمناقلة تلقائياً وبشكل آني وغير قابل للتجزئة. وباعتبار أننا في نظام موزع، فسوف تحتاج عملية التثبيت إلى مشاركة العديد من الإجرائيات الموزعة على حواسيب مختلفة، تحمل كل منها عدداً من المتحولات والملفات وقواعد البيانات والأغراض الأخرى التي تعدلها المناقلة. وسوف ندرس في هذا المقطع البروتوكول الأكثر انتشاراً لتنفيذ التثبيت الذري في نظام موزع، ويعرف باسم التثبيت على مرحلتين.



الشكل (10) التثبيت على مرحلتين

يظهر الشكل **10** عملية التثبيت على مرحلتين، حيث تعمل إحدى الإجرائيات كمنسق، وهي عادة الإجرائية التي تنفذ المناقلة. وتنفذ العملية بحسب الخطوات التالية:

1. تبدأ عملية التنفيذ عندما يكتب المنسق سطراً في السجل يذكر فيه أنه باشر عملية التثبيت.
2. يقوم المنسق بإرسال رسالة لكافة الإجرائيات المشاركة في المناقلة والتي تسمى بالتابعة **subordinates** يخبرهم فيها بالاستعداد لعملية التثبيت.
3. عندما تستلم إجرائية تابعة رسالة الاستعداد للتثبيت تتحقق من السجل لديها من جاهزيتها لذلك، وتضيف سطراً إلى السجل تذكر فيه قرارها وتعيد الإجابة إلى المنسق.
4. عندما يستلم المنسق كافة الإجابات يقرر التثبيت في حال كون كافة الأجوبة إيجابية (مستعد للتثبيت)، وفي حال وجود رفض من إحدى الإجرائيات أو عدم وصول إجابة فيتم إلغاء المناقلة.
5. في كلتا الحالتين يكتب المنسق قراره في السجل ويرسل رسالة إلى كافة الإجرائيات التابعة ليعلمها بالقرار. وتعتبر هذه الكتابة هي العملية النهائية التي تثبت المناقلة ولا

عودة عنها بعد ذلك بغض النظر عما يمكن أن يحصل لاحقاً.
نظراً لاستخدام التخزين المستقر لحفظ السجل فإن البروتوكول عالي الوثوقية ومقاوم للأعطـال.
ويترك للقارئ التحقق من صحة العمل في العديد من حالات التوقف:

- توقف المنسق بعد كتابة السجل الأول.
- توقف المنسق بعد كتابة نتيجة التصويت.
- توقف إجرائية تابعة قبل وبعد إجابتها على الرسالة الأولى.

المناقلات والتوارد

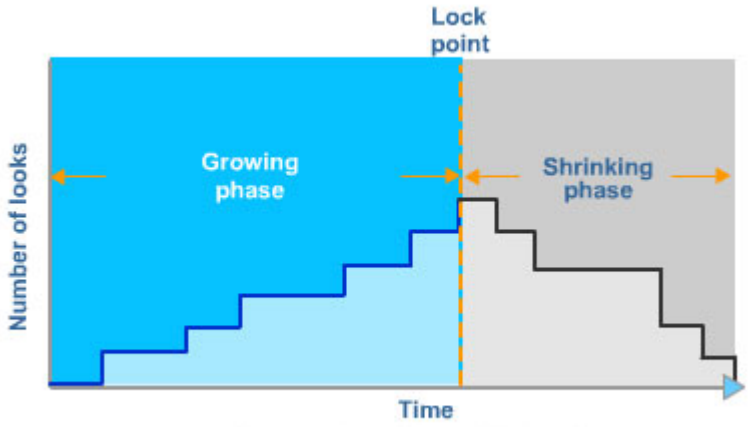
عندما تنفذ عدة مناقلات بشكل متوارد عبر إجرائيات عدة وعلى عدة معالجات، يحتاج النظام إلى تطبيق خوارزمية للتحكم بالتوارد **concurrency control algorithm** بهدف منع حصول تضاربات بين الإجرائيات. وسندرس في هذا المقطع ثلاث خوارزميات مختلفة.

الإقفال locking:

إن أقدم آلية للتحكم بالتوارد هي الإقفال، وهي في نفس الوقت أكثرها شيوعاً. فعندما تحتاج إجرائية إلى القراءة أو الكتابة إلى ملف (أو غرض آخر) كجزء من مناقلة، تقوم في البداية بوضع قفل على الملف بالاعتماد على مدير للأقفال **lock manager**. يمتلك مدير الأقفال قائمة بالملفات المقفولة، ويرفض كافة طلبات الإقفال الورادة والتي تخص ملفات أقفلت لصالح إجرائيات أخرى. ويقوم نظام إدارة المناقلات بطلب وتحرير الأقفال تلقائياً دون الطلب من المبرمج القيام بأي عمل.

يميز النظام بين نوعين من الأقفال لرفع درجة التوارد وتحسين الأداء، وتكون أقفال القراءة مشتركة، أي يمكن لأكثر من إجرائية وضع قفل للقراءة في آن واحد على نفس الملف، بينما لا يمكن وضع أي قفل آخر في حال وجود قفل كتابة على الملف (تعرف هذه الطريقة في إدارة السماحيات باسم طريقة **n-read, 1-write**).

نفترض لتسهيل الأمور أن الإقفال يتم على ملف بأكمله وليس أجزاء منه، وفي التطبيق العملي، تتغير حبيبة **grain** الإقفال بحسب نوع التطبيق، فيمكن أن تكون صغيرة بحيث يكون الإقفال على مستوى صفحة واحدة، أو كبيرة فيكون الإقفال على مستوى قاعدة بيانات بأكملها. وكلما صغرت حبيبة الإقفال كلما ازدادت درجة التوارد وتحسن الأداء، ولكنها تتطلب عدداً أكبر من الأقفال وبالتالي ترتبط بكلفة زمنية أكثر لوضع وتحرير الأقفال، كما يجب توخي الحرص لأنها قد تقود إلى استعصاءات.



الشكل (11) الإقفال على مرحلتين

تعتمد أغلب المناقلات على آلية إقفال تعرف باسم الإقفال على مرحلتين وتسمح بتلافي مشكلات عدم الاتساق والهروب من الاستعصاء، وهذه الآلية معروضة في الشكل 11.

تقوم الإجرائية في البداية بالاستحواذ على كافة الأقفال التي تحتاجها، وهذه هي المرحلة الأولى التي يزداد فيها عدد الأقفال باستمرار وبدون تناقص، ثم تدخل في مرحلة الانكماش والتي تقوم فيها بتحرير الأقفال دون أن تطلب الاستحواذ على أي قفل جديد. ونقبل بدون إثبات أن تطبيق هذه الخوارزمية يحقق خاصة وجود مكافئ متسلسل للمناقلات المتواردة **serializable**.

في كثير من المنظومات لا تبدأ المرحلة الثانية حتى انتهاء المناقلة وإجراء عملية التثبيت أو الإلغاء، وهو ما يعرف باسم الإقفال الصارم على مرحلتين **strict two-phase locking**. ولهذه المنهجية ميزتان:

- لن تقرأ أية مناقلة قيمة كتبتها مناقلة أخرى غير مثبتة، وبالتالي فلن يحصل هناك أي إلغاء لمناقلة بسبب اعتمادها على قراءة قيم ضمن ملف وألغيت تلك القيم (تعرف هذه الظاهرة باسم الإلغاءات المتتابعة **cascading aborts**).
- يمكن معالجة الاستحواذ على الأقفال وتحريرها تلقائياً من قبل النظام وبشكل ضمني **implicit** يريح مطور المناقلة من ذلك.

رغم ذلك يبقى هناك خطر حصول استعصاءات، فإذا حاولت إجرائيتان الحصول على زوج من الأقفال بترتيب متعاكس، فقد ينتج عن ذلك استعصاء، وتطبق هنا آليات تلافي الاستعصاء التقليدية، بما فيها إلغاء المناقلة في حال تجاوز الانتظار للحصول على قفل الحدود الطبيعية.

التحكم المتفائل بالتوارد:

هناك طريقة ثانية للتعامل مع مناقلات متواردة تعرف باسم التحكم المتفائل بالتوارد. والفكرة هنا بسيطة جداً: دعونا نفعل ما نشاء وسنعالج التضاربات عند حدوثها. تستند هذه الخوارزمية إلى نتائج عملية تظهر قلة حصول التضاربات، وبالتالي فإن هذه الخوارزمية تعمل جيداً في كثير من الحالات.

تحتفظ الخوارزمية بقائمة الملفات التي قامت المناقلة بقراءتها وبتعديلها، وعند لحظة تثبيت التنفيذ، تتحقق الخوارزمية من أن الملفات التي تتعامل معها لم تتغير منذ بداية المناقلة، وذلك عبر المقارنة مع قوائم الملفات الخاصة بالمناقلات الأخرى. ويتم التثبيت في حال عدم وجود تعديل، وإلا تلغى المناقلة وتعاد من جديد.

تتوافق هذه الخوارزمية إلى حد كبير مع التنفيذ المبني على فضاء العمل الخاص، إذ تسمح تلك الآلية بتلافي التضارب أثناء التنفيذ وحتى الوصول إلى لحظة التثبيت.

إن الميزة الكبرى لهذه الخوارزمية هو عدم وجود استعصاءات وكونها تعطي أكبر درجة توارد ممكنة لأن الإجرائيات لا تحتاج إلى الانتظار للحصول على أقفال.

أما مساوئها فهي مرتبطة بكون احتمال الفشل أكبر منه في الخوارزميات الأخرى، وبالتالي فهي غير مناسبة في المنظومات التي يكون فيها ضغط كبير على المنظومة وعدد كبير من المناقلات المتواردة حيث يتزايد احتمال التضارب إلى حد ملموس.

الأختام الزمنية:

هناك طريقة ثالثة مختلفة كلياً عن سابقتيها وتعتمد إلى حد كبير على خوارزمية لامبورت التي سبق ورأيناها في هذا الفصل.

ترتبط بداية المناقلة بختم زمني وحيد، ولكل ملف في النظام ختم زمني للقراءة وختم زمني للكتابة تحدد الختم الزمني لآخر مناقلة مثبتة قامت بقراءته وبتعديله. ويقتضى الترتيب الصحيح لعمل المناقلات ان يكون الختم الزمني الذي على الملف أصغر من الختم الزمني للمناقلة التي ترغب باستخدامه.

عندما لا يكون الترتيب صحيحاً، فهذا يعني أن هناك مناقلة بدأت بعد المناقلة الحالية، وقامت بالتعامل مع الملف، ومن ثم ثبتت أعمالها. وهذا يعني أن المناقلة الحالية متأخرة كثيراً ولن يكون بإمكانها المتابعة، وبالتالي يجب إلغاؤها. ويتم اكتشاف هذا الوضع عندما تحاول المناقلة التعامل مع ملف فتجد أن الختم الزمني على هذا الملف أكبر من الختم الزمني الخاص بالمناقلة.

وهذه الخوارزمية تشبه إلى حد ما الخوارمية التفاؤلية (نحن نعمل وإذا حصلت مشكلة نلغي المناقلة)، وتبقى أقل فاعلية من خوارزميات الإقفال لأنها تزيد كثيراً من احتمال الإلغاء ولكنها بالمقابل لا تعاني من مشكلة احتمال الاستعصاء، وهي ميزة كبيرة بحد ذاتها.

الفصل الخامس

أنظمة التشغيل الموزعة

1. مقدمة

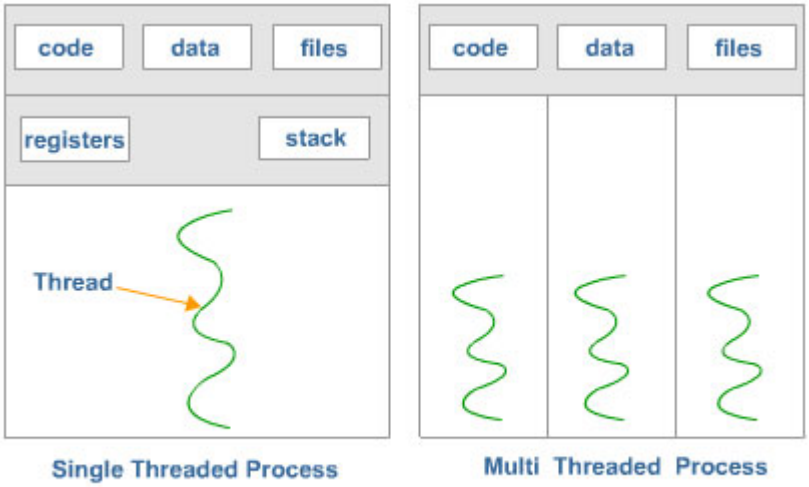
بعد أن درسنا العديد من الخوارزميات الهامة في النظم الموزعة، ننتقل في هذا الفصل إلى دراسة أنظمة التشغيل الموزعة بأجزائها المختلفة. وسنركز على عرض الاختلافات الأساسية بين مكونات نظام التشغيل الموزع وما يقابله في نظام تشغيل مركزي.

2. الإجرائيات الموزعة

تعتبر الإجرائيات هي واحدة التنفيذ التقليدية في الأنظمة التقليدية، ولكن ظهور الأنظمة الموزعة كان الخطوة الأولى التي دفعت مصممي أنظمة التشغيل نحو الانتقال إلى مفهوم آخر للتنفيذ، ألا وهو الاعتماد على النياسب **threads**. ونذكر بإيجاز بالمفهوم الأساسي للنيسب.

النياسب

ظهر مفهوم النيسب في نهاية الثمانينات، وكان يهدف بشكل أساسي إلى حل مشكلة بطء تبديل السياق التي تعاني منها الإجرائية نظراً لضخامة السياق المخصص لها، والتي تحد في النهاية من درجة التوارد المتاحة في النظام. بالإضافة إلى مشكلة صعوبة التخاطب بين الإجرائيات بسبب عزل فضاءات العنونة عن بعضها البعض. وكان الحل هو الفصل بين مفهومي التنفيذ وتخصيص الموارد، بحيث تبقى الإجرائية هي واحدة تخصيص الموارد، وتعرف واحدة جديدة للتنفيذ هي النيسب.



الشكل (1) النياسب والإجرائيات

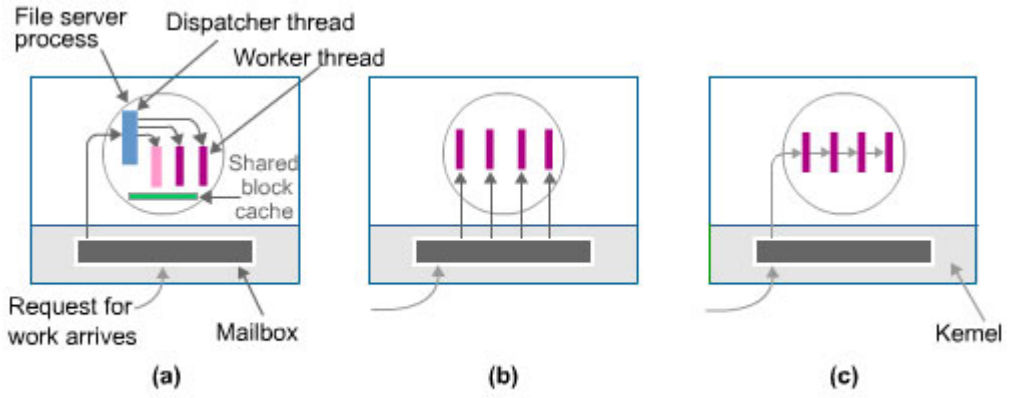
يمكن أن تحوي الإجرائية عدة نياسب تشترك كلها في الموارد المخصصة للإجرائية. لكل نيسب

سياق تنفيذ خاص به مكون من مجموعة سجلات المعالج وعداد البرنامج ومكدس. أما سياق الإجرائية فهو سياق الموارد، كصورة الإجرائية في الذاكرة والملفات المفتوحة لصالح الإجرائية (أي أن نظام التشغيل لا يعرف أي نيسب في الإجرائية طلب فتح الملف).

يظهر الشكل (1) الفرق بين النموذج القديم للإجرائية التي تحوي نيسباً واحداً، ونموذج الإجرائية المتعددة النياسب. ونلاحظ أن الرسم يظهر بوضوح وجود سياق خاص بكل نيسب، وسياق آخر مشترك بين كل نياسب الإجرائية.

استخدامات النياسب

تم إدخال النياسب لتحقيق خصائص التوارد والتنفيذ المتسلسل ضمن النيسب الواحد واستخدام استدعاء النظام التي تؤدي إلى التوقف. يظهر الشكل -2- مخدماً للملفات، حيث نرى أن يوجد لدينا نيسب أساسي وهو الموزع **dispatcher**، مسؤول عن استقبال الطلبات الواردة للتنفيذ، ومن ثم توجيهها إلى نيسب عامل **worker thread**، وإيقاظ النيسب المتوقف (مثال: تنفيذ عملية V على السيمافور الذي يستخدمه النيسب العامل للتوقف).



الشكل (2) استخدام النياسب في المخدم

عند استيقاظ النيسب العامل يتأكد من إمكانية تحقيق الطلب من الذاكرة الخابئة لوحدات القرص الصلب وهي مشتركة بين كافة النياسب العاملة. وإذا لم تكن موجودة يقوم النيسب بتنفيذ استدعاء من استدعاءات النظام يؤدي إلى توقفه عن العمل إلى حين وصول الوحدة المطلوبة. وحال توقف النيسب يقوم المجدول **scheduler** بتسليم المعالج إلى نيسب آخر قد يكون الموزع الذي يستلم طلباً جديداً أو قد يكون نيسباً عاملاً آخر.

كيف يمكن تنفيذ المخدمات في غياب النياسب؟ لو كان التنفيذ يعتمد على إجرائية وحيدة النيسب، فسوف يتوقف عمل المخدم في انتظار إجابة القرص الصلب، وبالتالي لن يكون من الممكن

الاستفادة من وحدة المعالجة بالحد الأعظمي. وبالتالي فمن الواضح أن استخدام النياسب يسمح يرفع نسبة الاستفادة من وحدة المعالجة.

يمكن أيضاً برمجة المخدم عن طريق إجرائية واحدة تنفذ أوتومات حالة منتهية وتستخدم استدعاءات غير معلقة لنظام التشغيل، ولكن العملية معقدة جداً. ويبقى استخدام مخدم متعدد النياسب هو الشكل الأفضل أداء وسهولة في البرمجة.

يعرض الجزء **b** من الشكل **-2-** طريقة أخرى لتنظيم الإجرائيات متعدد النياسب وهو نموذج الفريق **team**: وهنا تكون كافة النياسب متكافئة، وتحصل كل منها على الطلبات من رتل انتظار الطلبات وتقوم بتنفيذها مباشرة. أما الجزء **c** فهو يعرض نموذج سلسلة الإنتاج **pipeline** حيث يقوم كل نيسب بتوليد بعض البيانات وتمريرها للنيسب الذي يليه، وهو نموذج يتوافق مع مسائل المنتج / المستهلك أكثر منه مع مخدمات الملفات.

لايقتصر استخدام النياسب على المخدمات، وإنما يمكن أن تتواجد أيضاً في الزبائن، ومن أبسط الأمثلة تطبيقات الواجهة التخاطبية البيانية التي تعتمد النياسب بكثافة في عملها.

3. نماذج إدارة الموارد في النظام الموزع

إن أهم الموارد على الإطلاق في النظام الموزع هي المعالجات المتواجدة في النظام. وتطرح أنظمة التشغيل الموزعة سؤالاً هاماً: كيف يمكن الاستفادة من تعدد المعالجات؟

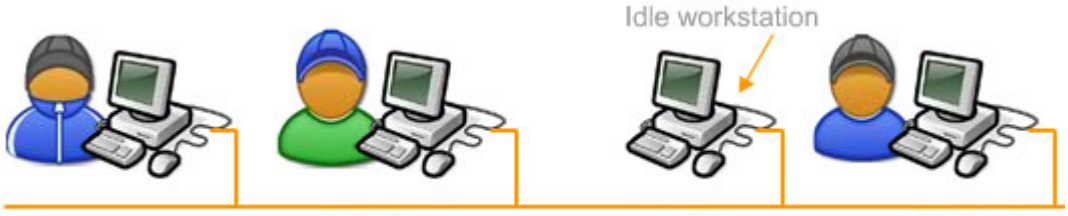
نعرض في هذا المقطع نموذجين أساسيين للتعامل مع تعدد المعالجات:

- نموذج محطة العمل **workstation**.
- نموذج المجمع **pool**.

نموذج محطة العمل

هذا النموذج بسيط جداً: يتكون النظام الموزع من عدد م محطات العمل الموزعة عبر بناء أو جامعة ومتصلة باستخدام شبكة محلية عالية السرعة، كما يظهر الشكل **3**. يمكن أن يكون بعضها متواجداً في مكتب خاص وبالتالي يستخدمها شخص واحد أو في صالة للعموم ويتعاقب عليها عدة مستخدمون. وفي كلتا الحالتين، فإن لمحطة العمل مستخدما واحداً في لحظة ما.

يمكن أن تكون المحطات بدون أقراص تخزين، وفي هذه الحالة تعتمد على مخدمات متخصصة بتقديم خدمة الملفات. وهو خيار شائع لأسباب لها علاقة بسهولة الإدارة والصيانة وانخفاض الكلفة.



الشكل (3) نموذج محطات العمل

أما إذا كانت المحطات مزودة بأقراص صلبة، فيمكن استخدامها ضمن أربعة سويات على الأقل:

1. ذاكرة افتراضية **paging** وملفات مؤقتة.
2. ذاكرة افتراضية وملفات مؤقتة وملفات النظام.
3. ذاكرة افتراضية وملفات مؤقتة وملفات النظام وذاكرة خابئة للملفات.
4. نظام ملفات كامل.

Disk usage	Advantages	Disadvantages
(Diskless)	Low cost, easy hardware and software maintenance, symmetry and flexibility	Heavy network usage: file servers may become bottlenecks
Paging. scratch files	Reduces network load over diskless case	Higher cost due to large number of disks needed
Paging. scratch files. binaries	Reduces network load even more	Higher cost ;additional complexity of updating the binaries
Paging.scratch files,binaries, file caching	Still lower network load;reduces load on file servers as well	Higher cost; cache consistency problems
Full local file system	Hardly any network load; eliminates need for file servers	Loss of transparency

الشكل (4) الأشكال المختلفة لاستخدام الأقراص الصلبة في محطات العمل

يظهر الشكل -4- مقارنة بين المحطات بدون أقراص والسويات المختلفة للمحطات المزودة بالأقراص، وهي مرتبة من الأعلى إلى الأسفل بحسب تناقص اعتمادها على مخدم الملفات.

ويمكن تلخيص مزايا نموذج محطات العمل كما يلي:

- النموذج سهل وواضح.
- لدى المستخدمين كمية محددة وثابتة من "طاقة الحوسبة"، وبالتالي لديهم زمن استجابة شبه مضمون.
- يتمتع المستخدم باستقلالية عالية، وتزداد هذه الاستقلالية لدى وجود قرص صلب في المحطة.

ويعاني النموذج من مشكلتين أساسيتين:

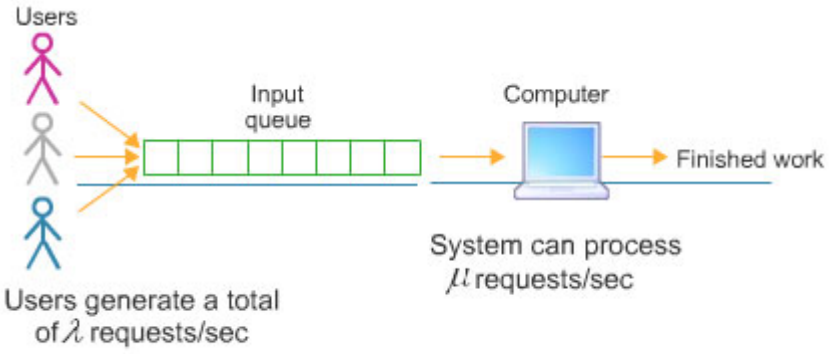
- عدم استغلال المساحة بشكل فعال، إذ لا تحوي محطات العمل عادة على أكثر من معالج واحد (وهو أمر تنبه إليه مصنعو المعالجات الصغرية وبدؤوا في تداركه مع المعالجات الجديدة من نوع **multicore** والتي تتضمن عدة معالجات في جسم واحد).
- هناك الكثير من الوقت الضائع إذا لاتستغل المحطة إلا عندما يكون المستخدم أمامها. ويصعب وضع موارد المحطة تحت تصرف المستخدمين الآخرين.

نموذج المجمع

يهدف هذا النموذج إلى معالجة مشكلة سوء استغلال الموارد الذي يعاني منه نموذج محطات العمل.

ويعتمد هذا النموذج على بناء مجمع من المعالجات (مثال: خزانة ممتلئة بالمعالجات متوضعة في مركز حساب)، يمكنها أن تخصص ديناميكياً لتخديم طلبات المستخدمين. وعوض أن يكون لدى المستخدم محطة عمل كاملة، سيعمل على مطراف **terminal** مزود بواجهة تخاطبية بيانية عالية الأداء.

إن هذا النموذج أقرب بكثير إلى الحوسبة بتشارك الزمن **timesharing** من مفهوم الحاسوب الشخصي الذي بني أصلاً عليه. وهو يعتمد على دفع فكرة المحطة بدون قرص صلب إلى مرحلة إضافية: تجميع كافة العناصر ذات الكلفة المرتفعة في المخدمات، وبالتالي يصبح نموذج المجمع وكأنه "مخدم معالجة".



الشكل (5) النموذج الأساسي لنظرية الأرتال

ويعتمد النموذج في بنائه على نظرية الأرتال **queuing theory**، وهي نظرية معروفة ومدروسة ويعرض الشكل -5- المكونات الأساسية لهذه النظرية. ومن أهم فوائد الاعتماد عليها هي إمكانية نمذجة هذا النوع من الأنظمة وتحليل أدائها بالاعتماد على طرق ومنهجيات رياضية معيارية.

ويمكن إثبات أن زمن الاستجابة الوسطي (الزمن الفاصل بين إرسال الطلب والحصول على الإجابة) يمثل بالعلاقة:

$$T = \frac{1}{\mu - \lambda}$$

وبدون الدخول في حسابات معقدة، من البديهي أن معدل إرسال الطلبات يجب أن يكون أقل من عدد الطلبات التي يمكن للمخدم معالجتها في الثانية وإلا لنما رتل الإنتظار إلى ما لا نهاية.

يمكن أيضاً استنتاج العلاقة التالية:

$$T_1 = \frac{1}{n\mu - n\lambda} = T / n$$

والتي تظهر أنه في حال استبدال **n** معالج صغير بمعالج قوته **n** ضعف قوة المعالج الواحد فإننا سنحصل على زمن استجابة أفضل بـ **n** مرة. وهي نتيجة مفاجئة إذ أنها قد تفهم على أنها معاكسة لمبدأ الحوسبة الموزعة إذ أنها تنصح بالاعتماد على حواسيب مركزية قوية أكثر من الاعتماد على محطات عمل ضعيفة وموزعة.

ورغم صحة هذه العلاقة رياضياً لكن يجب توخي الحذر عن إسقاطها على أرض الواقع، فليس بالضرورة أن يكون ممكن تكنولوجياً تصنيع معالج أقوى بـ **50** مرة مثلاً من معالج محطة العمل. ناهيك عن زيادة الوثوقية والقدرة على مقاومة الأعطال التي نحصل عليها عند وجود عدة معالجات.

4. أنظمة الملفات الموزعة

إن أحد أهم مكونات النظام الموزع هو نظام الملفات ومن الطبيعي أن التطبيقات الأولى للأنظمة الموزعة التي تهدف أساساً لتوسيع التشارك في الموارد كانت في مجال أنظمة الملفات. وتشبه مهمة نظام الملفات في نظام موزع تلك التي يقوم بها في نظام عادي، إذ يكون مسؤولاً عن تخزين البرمجيات والمعطيات وإتاحة الوصول إليها عند الطلب. وبالتالي لن نقوم بتكرار تفاصيل هذه المهام الأساسية هنا، وإنما سنركز على الخصوصيات المتعلقة بالأنظمة الموزعة.

نبدأ هنا بالتمييز بين مفهومين أساسيين: خدمة الملفات **File service**، ومخدم الملفات **File server**. فخدمة الملفات هي عبارة عن مجموعة الخدمات التي يقدمها نظام الملفات للزبائن (التطبيقات التي تحتاج إلى خدمة الملفات). وهي توصف الخدمات المتاحة والوسائط المستخدمة في استدعاءات هذه الخدمات، ولكنها لا تحدد أبداً كيف يتم تحقيق هذه الخدمات. أما مخدم الملفات فهو عبارة عن إجرائية تعمل على أحد الحواسيب وتسمح بتحقيق خدمة الملفات، ويمكن أن تتضمن خدمة الملفات أكثر من مخدم يتعاونون سوية على تقديم الخدمة. ولا يعرف الزبائن ما هو عدد المخدمات وأين تتوضع، فكل ما يعرفونه أن عليهم استدعاء إجرائية ما من مخدم

الملفات للحصول على النتيجة المرجوة. وفي أفضل الحالات، لا يعرف الزبون حتى أن نظام الملفات هو موزع بل يعطيه النظام الانطباع بأنه مركزي.

يسمح تصميم نظام الملفات على شكل إجرائية بوضع عدة مخدمات على نفس النظام يقوم كل منها بتقديم خدمة مختلفة لأنظمة الملفات. فعلى سبيل المثال يمكن أن يتضمن نظام الملفات مخدماً يقدم خدمة الملفات التقليدية لنظام **Unix**، ومخدماً آخر يدعم نظاماً افتراضياً يدعى **CFS** يقوم بضغط الملفات على الحارك، ومخدماً ثالثاً يدعم نظام ملفات **NTFS**. وتقوم خدمة الملفات بتوجيه الطلبات إلى المخدم المطلوب بما يتوافق مع طلب الزبون.

تصميم أنظمة الملفات الموزعة

يتضمن نظام الملفات الموزع عادة مكونين أساسيين:

- خدمة الملفات، وهي معنية بتنفيذ العمليات على الملفات كالقراءة والكتابة والإضافة.
- خدمة الأدلة **directory** وهي معنية بتنسيق الملفات وتوضعها ضمن النظام وإنشاء المجلدات وإضافة وحذف الملفات منها.

خدمة الملفات:

السؤال الأول المطروح في خدمة الملفات، وبغض النظر عن كونها مركزية أو موزعة هو التالي: ما هو الملف ؟ ففي العديد من أنظمة التشغيل مثل **UNIX** و **MS-DOS**، يعتبر الملف هو سلسة مستمرة من الثمانيات **bytes**، ويقع عاتق تفسير تلك الثمانيات كلياً على التطبيق بدون أن يكون لنظام التشغيل أية علاقة بذلك. وباعتبار أن أغلب الأنظمة الموزعة مبنية حول بيئة **UNIX** أو **Windows** الشبيهة بـ **MS-DOS**، فإن أغلب مخدمات الملفات تتعامل بدورها مع الملفات على أنها سلسلة من الثمانيات.

يوجد للملف العديد من الخصائص **Attributes**، وهي مجموعة من المعلومات عن الملف وليست جزءاً من الملف بحد ذاته. والخصائص الأكثر شيوعاً هي هوية مالك الملف والحجم وتاريخ الإنشاء وتاريخ التعديل وسماحيات التعامل مع الملف. ويمكن أن تتضمن خدمة الملفات استدعاءات تسمح بتعديل جزء من هذه الخصائص كحقوق الكتابة. ولكن لا يمكن مثلاً تعديل تاريخ الإنشاء أو حجم الملف (إلا عن طريق إضافة أو حذف بيانات من الملف).

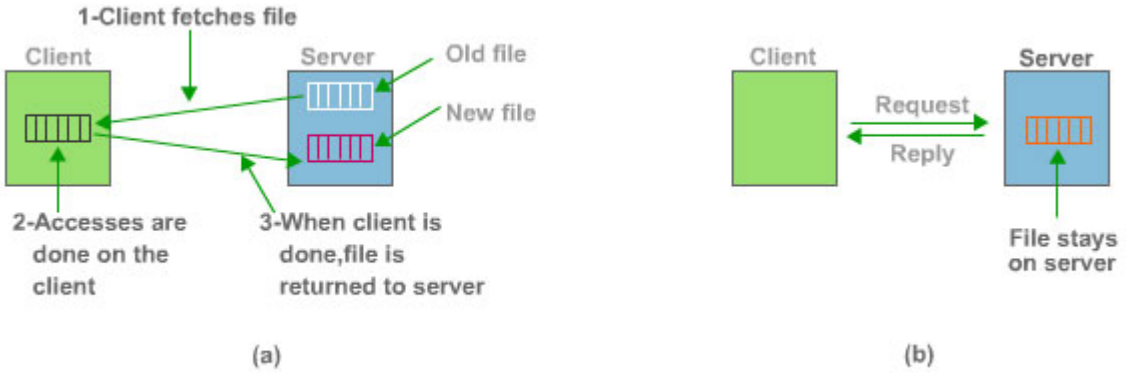
ومن الخصائص الهامة في الأنظمة الموزعة هي وجود ملفات غير قابلة للتعديل **immutable**. أي أنه لا يمكن تعديل الملف بعد إنشائه. وبالتالي يسهل دعم وجود نسخ متعددة من نفس الملف بهدف تسريع النفاذ إليه، كما يسهل دعم وجود نسخة خابئة **cache** من الملف وذلك بسبب انتفاء

مشكلة تعديل النسخ المتعددة للملف عندما يكون هناك أي تعديل.

أما الحماية في الأنظمة الموزعة فهي تشبه إلى حد كبير تلك الموجودة في الأنظمة وحيدة المعالجات: قوائم السماحيات **ACL** والإمكانيات **capabilities**. فبحسب آلية الإمكانيات يكون لدى كل مستخدم تذكرة نسميها إمكانية، تحدد ماهي السماحيات الخاصة به لكل غرض من الأغراض التي يمكنه الوصول إليها.

أما قائمة السماحيات فهي تحدد لكل ملف قائمة المستخدمين الذين يحق لهم الوصول إلى الملف وما هي حقوق كل منهم، وهي تشبه في ذلك قائمة السماحيات الموجودة في نظام **UNIX**.

تنقسم خدمات الملفات إلى نوعين: النوع الأول يعمل بحسب مبدأ التحميل والتنزيل **upload/ download** وفيه تقتصر الخدمات التي يقدمها نظام الملفات على اثنتين فقط: قراءة ملف وكتابة ملف. وتقوم عملية قراءة الملف بنسخ الملف بأكمله من مخدم الملفات إلى الزبون الذي يطلب الملف. أما عملية الكتابة فتقوم بنسخ الملف بأكمله في الاتجاه العكسي (من الزبون إلى المخدم). أي أنه لا يمكن نقل أجزاء من ملف بل أن الزبون سيقوم دوماً بنسخ الملف بأكمله ولو كان بحاجة لقراءة بضعة أحرف منه. ويمتاز هذا النوع من المخدمات الموضح في الشكل رقم 1 **(a)** ببساطته التصميمية. إذ يقتصر عمل الزبون على استحضار نسخة من الملف واستخدامها محلياً، وفي حال وجود تعديلات على الملف ينسخ الملف بأكمله على المخدم. وتكمن نقطة الضعف الأساسية لهذا النوع في الحاجة إلى مساحة تخزين كبيرة على الزبون لتخزين الملفات المنقولة من المخدم، كما أن نسخ الملف بأكمله لقراءة جزء صغير منه قد يمثل هدراً ملموساً للوقت ولسعة التخزين لاسيما في حال كون الملف كبيراً (جداً).



الشكل (6) الأنواع المختلفة لمخدمات الملفات

النوع الآخر موضح في الشكل رقم **6 (b)**، ويعرف باسم النفاذ البعيد **remote access model**. وفي هذا النوع من المخدمات تقدم خدمة الملفات عدداً كبيراً من العمليات لفتح وإغلاق الملفات

وقراءة وتعديل أجزاء من الملفات، التنقل ضمن الملف، وفحص وتغيير خصائص الملفات.
وبالمقارنة من النوع الأول الذي يتم فيه تنفيذ كافة العمليات على الزبون بعد نسخ الملف، فإن التنفيذ هنا يتم على المخدم. والميزة الأساسية لهذا النوع هي في إلغاء الحاجة لنقل الملف بأكمله وبالتالي اختصار الوقت ومساحة التخزين اللازمة للتعامل مع الملف.

خدمة الأدلة:

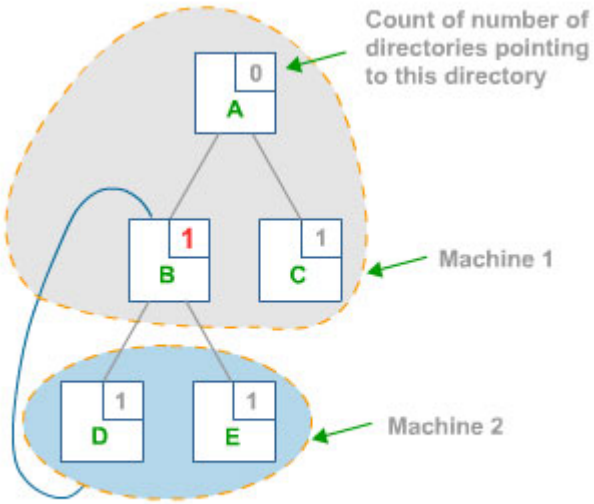
إن الجزء الآخر من خدمة الملفات هو خدمة الأدلة، وتقدم عمليات إنشاء وحذف الأدلة وتسمية الملفات وتغيير أسمائها، ونقل الملفات من دليل لآخر. ولا علاقة بخدمة الأدلة بنوع خدمة الملفات المذكورة في البند السابق (هل تنسخ الملفات بأكملها أم يتم النفاذ إليها عن بعد).

تعرف خدمة الأدلة عدد من المحارف والصيغ المستخدمة في تركيب أسماء الملفات والأدلة. وتتكون أسماء الملفات عادة من حرف واحد إلى حد أعلى ما من المحارف والأرقام وبعض المحارف الخاصة. وتقسم بعض الأنظمة أسماء الملفات إلى جزئين عادة ما يكونان مفصولين بنقطة ("."), وبحيث يستخدم الجزء الثاني للتعرف على نوع الملف، مثل الاسم **procedure.c** الذي يمثل ملفاً مكتوباً بلغة **C**، أو **nothing.txt** الذي يمثل ملفاً نصياً عادياً. وتشبه الأدلة الموزعة الأدلة المركزية في كونها تسمح بوجود الأدلة الجزئية **subdirectories** وكافة العمليات المتعلقة بها مما يجعل الدليل مكوناً في الحقيقة من شجرة من الأدلة ضمن ما يعرف باسم نظام الملفات الهرمي **hierarchical file system**.

ويمكن في بعض الأنظمة أن يتم إنشاء روابط أو مؤشرات نحو دليل ما، مما يجعل من نظام الأدلة بياناً عشوائياً وهو شكل أكثر غنى بالإمكانيات من الشجرة، وتزداد أهمية هذا الشكل في الأنظمة الموزعة. ويظهر الشكل 7 دليلاً موزعاً على ثلاثة حواسيب مختلفة، كما يمكن رؤية المشكلة التي تنتج عن وجود المؤشرات والروابط العشوائية.

يتضمن الدليل **D** مؤشراً نحو الدليل **B**، وتظهر المشكلة عندما يتم حذف الرابط بين **A** و **B**. ففي حالة الاقتصار على البنية الشجرية، لا يمكن حذف الرابط الخاص بدليل إلا عندما يكون الدليل خاوياً. أما في حالة البيان ذو البنية العشوائية، فإنه من المسموح حذف الرابط عندما يكون هناك رابط آخر على الأقل يشير إلى الدليل **B**. ويتم الاعتماد على وجود عداد للمؤشرات، بحيث يمكن معرفة إذا ما كان الرابط المطلوب حذفه هو الأخير. وبالتالي فإن حذف الرابط من **A** إلى **B** هو مقبول نظرياً إذ ستبقى قيمة العداد **1**، ولكن سينتج عنه فعلياً أن الدليل **B** أصبح غير قابل للوصول إليه، وأن هذا الدليل وكافة الأدلة المتضمنة داخله (أي **D** و **E**) هي يتيمة

.orphaned

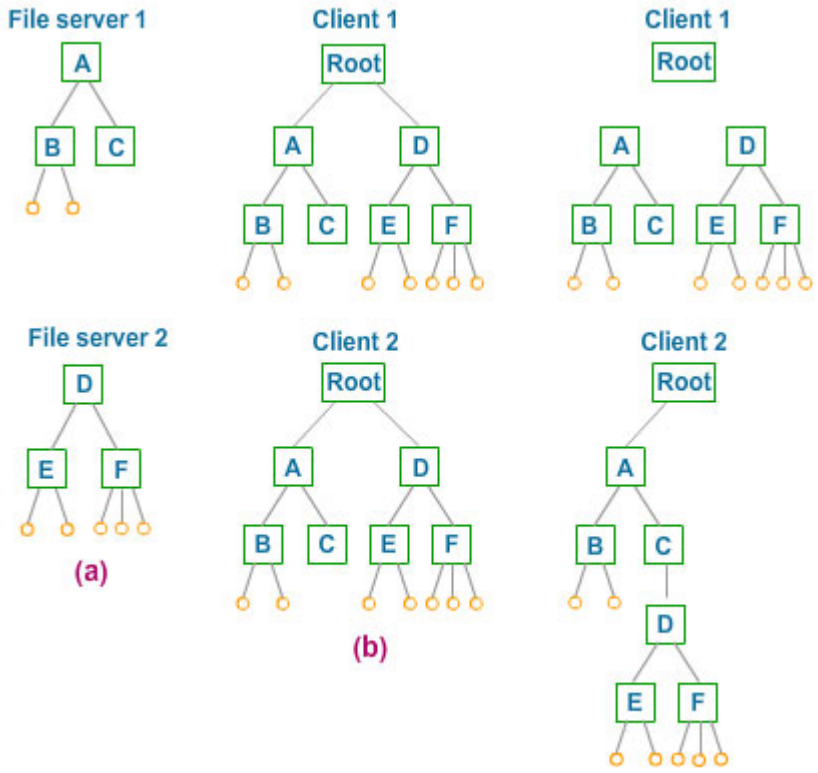


الشكل (7) بيان الاحتواء العشوائي للأدلة

هذه المشكلة موجودة أيضاً في الأنظمة المركزية، ولكن تزداد حدتها في الأنظمة الموزعة. إذ يمكن في حالة النظام المركزي اكتشاف الأدلة اليتيمة ولو أن هذه العملية مكلفة بعض الشيئ، وذلك لأن كافة المعلومات موجودة في مكان واحد ومن الممكن إيقاف كافة العمليات المتعلقة بنظام الملفات بهدف استكشاف بيان الأدلة. أما في حالة نظام موزع فمن الصعب بمكان (إن لم يكن مستحيلاً) إيقاف كافة العمليات دفعة واحدة.

تسمية الملفات Naming

إن أحد أهم الأسئلة المطروحة في حالة الأدلة هو ما إذا كان يتوجب أن يكون لكل الزبائن نفس الرؤية للبنيان الهرمي للدليل. ويظهر الشكل رقم 3 مثالاً توضيحياً لهذا السؤال. ففي الجزء a من ذلك الشكل نرى مخدمين للملفات يحوي كل منهما ثلاثة أدلة وعدداً من الملفات. وفي الجزء b نرى نظاماً يؤمن لكافة الزبائن نفس الرؤية لنظام الملفات الموزع، فإذا كان المسار /D/E/x صحيحاً على أحد الحواسيب، فسيكون صحيحاً على كافة الحواسيب. بينما يظهر الجزء c حالة مختلفة يمكن فيها أن يكون المسار صحيحاً على الزبون الأول ولكن ليس على الزبون الثاني. وهذه الحالة الأخيرة هي المعيارية في حالة المنظومات التي تعتمد على استخدام عدة أنظمة ملفات عن طريق التعليق عن بعد remote mounting. وهي الطريقة الأكثر مرونة والأسهل تحقيقاً، ولكنها تختلف عن السلوك المعياري الذي تعود عليه مستخدمو الأنظمة التقليدية.



الشكل (8) أشكال مختلفة لتسمية الملفات

دلالة التشارك في الملفات

إن التشارك في الملفات هو أحد أهم النقاط الواجب دراستها في الأنظمة الموزعة، لا سيما عند الحديث عن مخدمات الملفات التي تستقبل طلبات من كل الزبائن الممكنة. وبالتالي يجب تعريف دلالة القراءة والكتابة بدقة بهدف تلافي المشكلات التي يمكن أن تحصل.

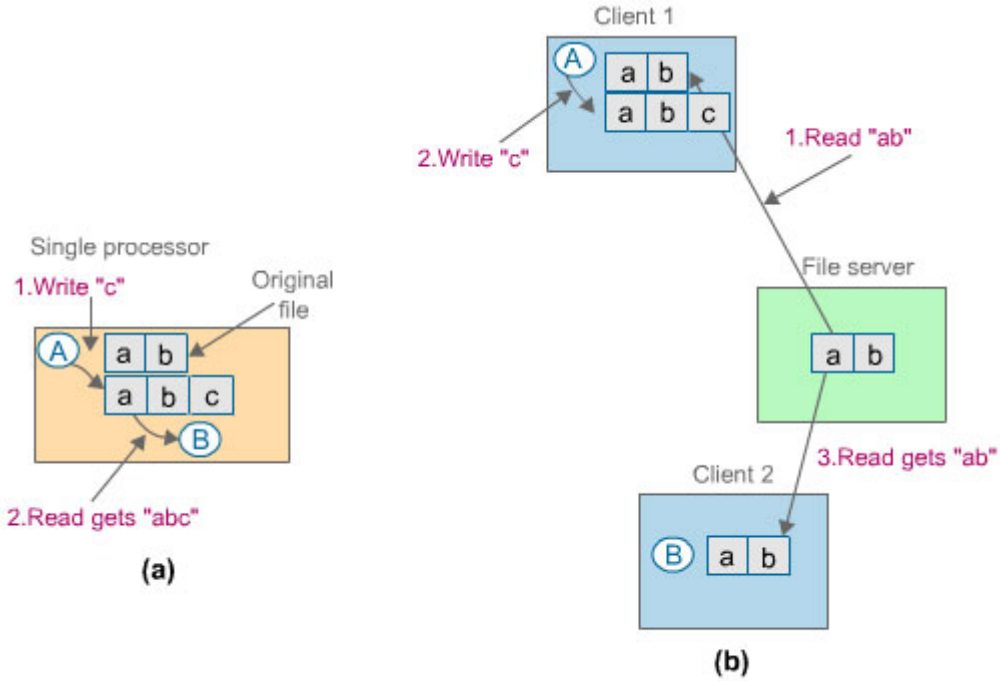
دلالة UNIX

في نظام مركزي مثل **UNIX** يضمن نظام الملفات الخاصة التالية:

- القيمة التي تعيدها عملية **READ** هي القيمة التي كتبتها آخر عملية **WRITE**.

إن تحقيق هذه الخاصة يفترض ضمنياً أن نظام التشغيل قادر على فرض ترتيب كلي على كل العمليات على ملف ما.

وتعرف هذه الخاصة باسم دلالة **UNIX** أو **UNIX semantics**، وهي معروضة في الشكل **9**.



الشكل (9) التشارك في ملف ضمن نظام مركزي مثل UNIX

دلالة الجلسة في النظام الموزع

من الممكن تحقيق هذه الخاصة في نظام مركزي فقط طالما كان عندنا مخدم واحد للملفات ولا توجد ذاكرة خابئة للملفات عند الزبون. في هذه الحالة ستذهب كافة الطلبات إلى نفس المخدم الأمر الذي يفرض تلقائياً علاقة الترتيب الكلي المنشودة.

ولكن هذه الشروط تحد كثيراً من أداء مخدمات الملفات، إذ إن الذاكرة الخابئة للزبون تسمح بتحسين الأداء إلى حد كبير. وتكمن المشكلة في أثر قيام الزبون بتعديل نسخته المحلية من الملف والموجودة في ذاكرته الخابئة. ففي هذه الحالة سيحصل طالب القراءة على قيمة خاطئة كما يظهر في الشكل **9-b**.

الحل الأول يكمن في نسخ التعديلات مباشرة على النسخة الأصلية، ولكنه يؤدي إلى تخفيض الأداء إلى حد كبير.

الحل المقترح يكمن في قبول خاصة أقل صرامة من تلك التي تعطيها الأنظمة المركزية:

لايمكن رؤية تعديلات الملفات المفتوحة إلا من قبل الإجرائيات التي قامت بتعديلها، ولا تصبح هذه التعديلات مرئية للإجرائيات الأخرى إلا بعد إغلاق الملف.

تعرف هذه القاعدة باسم دلالة الجلسة **session semantics** وهي نسخة مخففة من **UNIX semantics**.

ويطرح هنا السؤال التالي: ماذا يحصل في حال محاولة أكثر من زبون تعديل نفس الملف ؟ والجواب الأكثر بساطة هو أن الإجرائية التي تغلق الملف أخيراً هي التي ستفرض تغييراتها في النهاية، ويمكن أن تترك النتيجة بدون تعيين على أن تكون إحدى الإجرائيات هي التي ستفرض تغييراتها بأكملها دون أن تكون بالضرورة الأخيرة.

الملفات الصماء أو غير القابلة للتغيير

هناك طريقة أخرى مختلفة كلياً لمعالجة هذه المشكلة ألا وهي اعتماد ملفات غير قابلة للتغيير، بحيث لا يمكن أصلاً تغيير ملف قائم، وإنما ينشأ ملف جديد مكانه، والعمليات الوحيدة التي يتيحها مخدم الملفات هي **READ** و **CREATE**. تعرف هذه الآلية باسم الملفات غير القابلة للتغيير **Immutable files**.

وفي حال اعتماد هذا الحل يبقى السؤال قائماً عما يجب أن يحصل عند محاولة عدة إجرائيات تغيير ملف في نفس الوقت، إذ يعني ذلك أننا سنحاول إنشاء عدة ملفات عوضاً عن الملف القديم. والجواب هنا يشبه ما يحصل في حالة دلالة الجلسة، إذ نفترض أن أحد الملفات هو الذي سيحل محل الملف القديم، دون أن يكون بالضرورة الملف القديم.

المناقلات والملفات

يمكن أيضاً الاستعانة بالمناقلات التي سبق ورأيناها لضمان اتساق محتوى الملفات عند التشارك فيها. ونذكر بأنه في حال اعتماد المناقلات على إقفال من مرحلتين، فإن تنفيذ عدة مناقلات سيكون مكافئاً لترتيب تلسلي ما لهذه المناقلات واحدة تلو الأخرى وبدون تداخل. كما أن التغييرات تحقق خاصة هامة **All-or-nothing**، أي أن التغييرات على الملفات ستبقى غير مرئية حتى تقوم المناقلة بإنهاء عملها بنجاح وتثبت التعديلات عبر تعليمة **COMMIT**.

Method	Comment
UNIX semantics	Every operation on file is instantly visible to all processes
Session semantics	No changes are visible to other processes until the file is closed
Immutable file	No updates are possible; simplifies sharing and replication
Transactions	All changes have the all-or-nothing property

ويظهر الجدول الطرق المختلفة لإدارة موضوع التشارك في الملفات ضمن أنظمة التشغيل الموزعة وخصائص كل منها.

المراجع

A. Tanenbaum, Introduction to distributed systems.

J. Wu, Distributed systems design

Coulouris - Distributed Systems - Concepts and Design 3e.

A. Tanenbaum, Modern Operating systems, 2nd Edition.

Ajay D. Kshemkalyani, Mukesh Singhal,

Distributed Computing Principles, Algorithms, and Systems.